Back Propagation, the Easy Way (part 1)

Simple detailed explanation of the back propagation

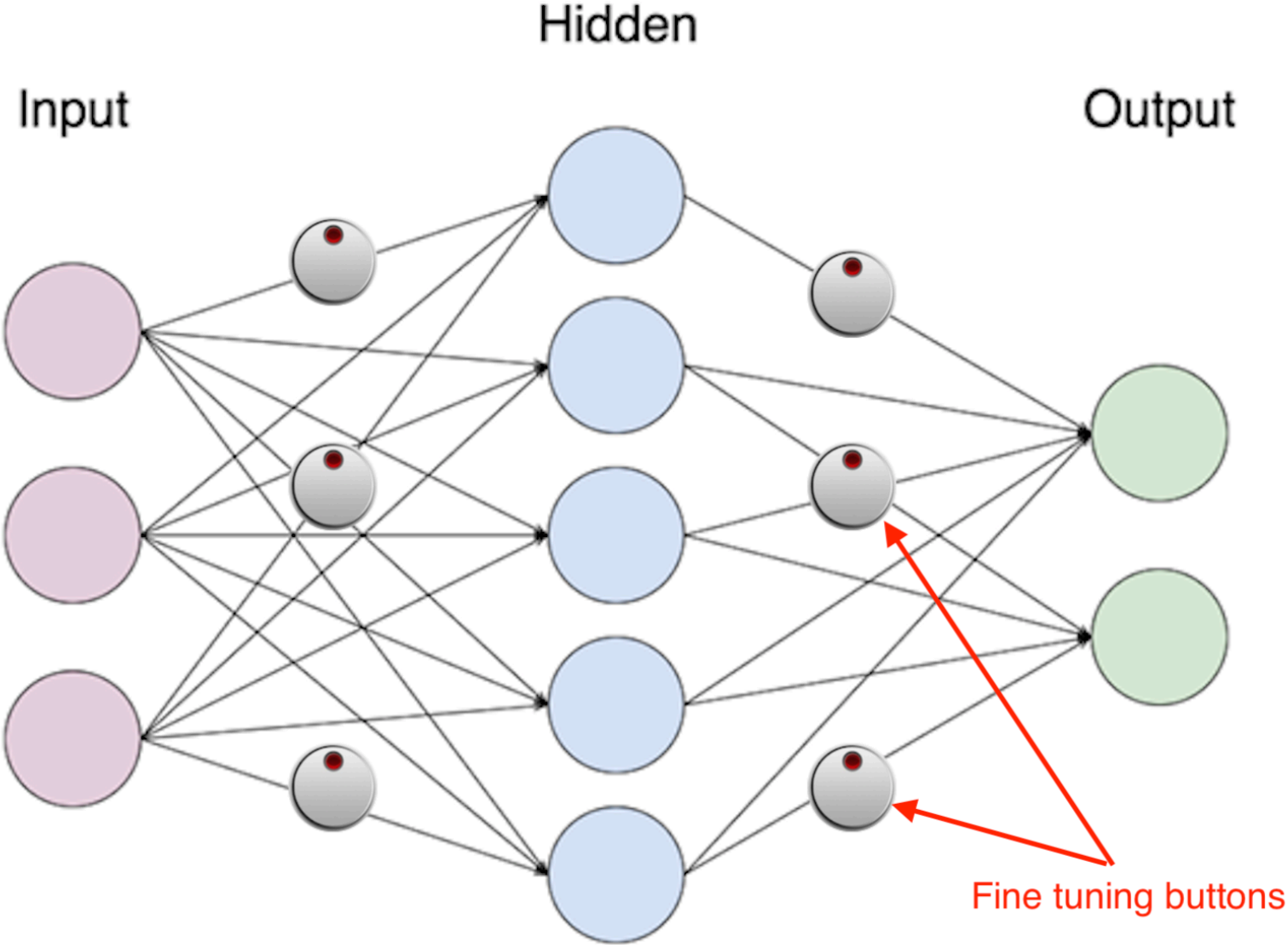
Ziad SALLOUM
Jan 5, 20198 min read

Update: The best way of learning and practicing Reinforcement Learning is by going to http://rl-lab.com

Following the article on Gradient Descent, the Back Propagation makes good use of this technique in order to compute the "right" values of weights in a neural network.

Important: This article contains a considerable number of equations, however they are all logically connected. The reader should have some patience going through them, and understand how they are built. It is essential to the good comprehension of the Back Propagation technique.

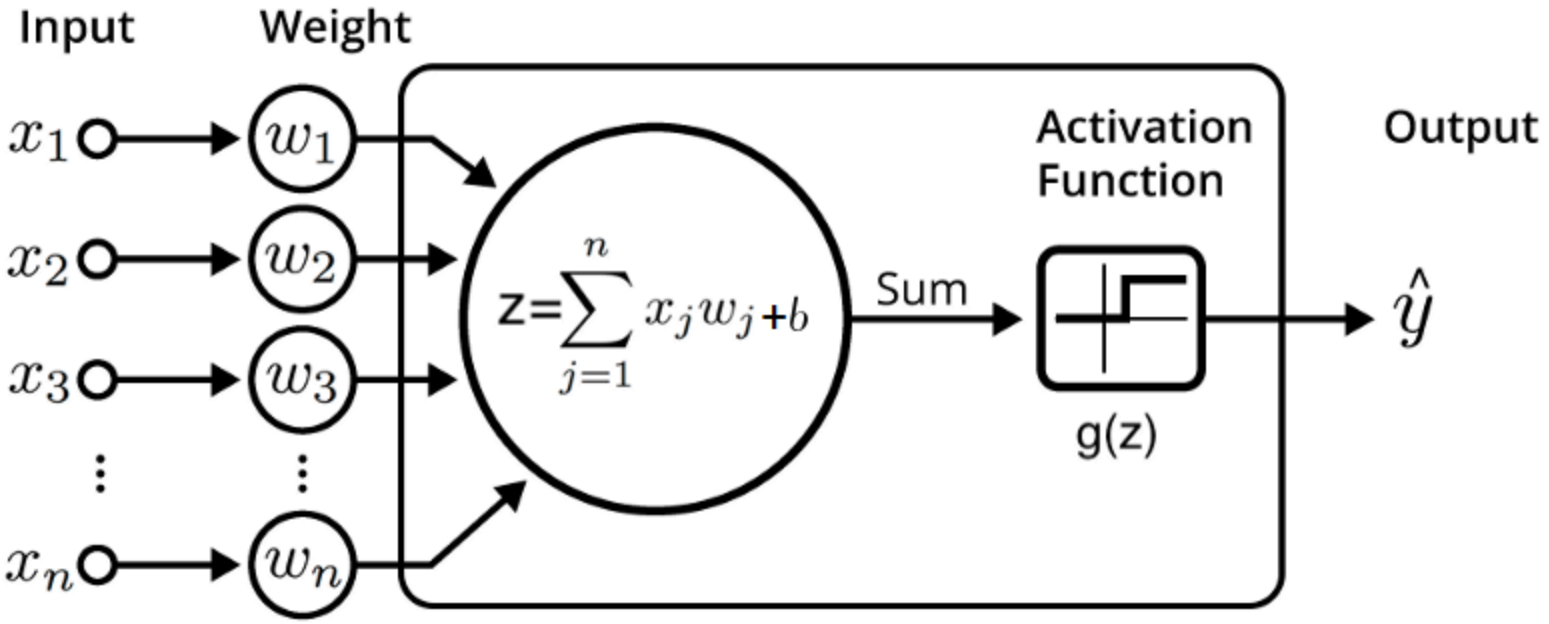
As with the Gradient Descent approach, we borrow the radio metaphor again to help understand the intuition of the Back Propagation. Whereas in the other article there was only one button to handle in order to get an acceptable output, in here we have many buttons that need adjusting.



The image above shows that there is a fine tuning button on each line connecting two nodes (even though they are not all represented).
As seen before, our job is to adjust those buttons so that the error at the output is minimum, which means that the effective output is as close as possible to the expected results.
In the Gradient Descent we talked about rather a simple relation between one fine tuning parameter θ and the error or loss $\mathcal{L}$, in this case we have a more complex relation.
However we will start slowly by considering only one neuron.

Case of One Neuron

As a reminder a single neuron receives one or more inputs ***x*** and multiplies each of the input by a certain weight ***w,*** sums them and adds a bias ***b***, **** then applies an activation function ***g()*** to produce an output **** $\hat{y}$.



To be able to assess the output of the neuron, a loss function $\mathcal{L}$ is used. One such loss function could be $\mathcal{L}(y, \hat{y}) = ½(y - \hat{y})^2$

The aim, of course, is to determine values of ***w*** and ***b*** in order to minimize $\mathcal{L}$.

By looking at the neuron architecture we can easily notice the following ***chain of events***: Variation of w_i causes ***z*** to vary, variation of ***z*** causes ***g(z)*** to vary, variation of ***g(z)*** causes $\mathcal{L}$**(y, ŷ)** to vary.

For simplicity of the notation let's call ***a = g(z)***, it follows that ***ŷ = a.*** The above chain reaction can be derived using the chain rule which states that the variation of the output relative to the $\mathbf{w_i}$ is the multiplication of the partial derivative of the phases in between:

$\partial\mathcal{L}/\partial w_i = \partial\mathcal{L}/\partial a \; \partial a/\partial z \; \partial z/\partial w_i$

Let's consider each term alone:

$\partial\mathcal{L}/\partial a = \partial(½(y - \hat{y})^2)/\partial a = \partial(½(y - a)^2)/\partial a = -(y - a) = a - y$

$\partial a/\partial z = \partial g(z)/\partial z =$ **g'(z)** (where g'(z) is the derivative of g(z). Note that g(z) can be any derivable function)

$\partial z/\partial w_i = \partial(x1\,w1 + x2\,w2 + \ldots + x_i\,w_i + \ldots xn\,wn)/\partial w_i = \partial(x_i * w_i)/\partial w_i = x_i$

Replacing each term by its value will give us the $\partial\mathcal{L}/\partial w_i$ $\partial\mathcal{L}/\partial w_i = (a - y)\; g'(z)\; x_i$

Let's define δ such as $\delta = (a - y) * g'(z)$, the final form of $\partial\mathcal{L}/\partial w_i$ will be:

$*\partial\mathcal{L}/\partial w_i \_= \delta\, x_{i\_}**$

If you recall from the Gradient Descent article, in order to find the θ that minimizes $\mathcal{L}$ we iterate the following equation until $\Delta\mathcal{L}$ becomes too small or zero: $\theta n_{+1} = \theta n - \alpha \,.\, \Delta\mathcal{L}$

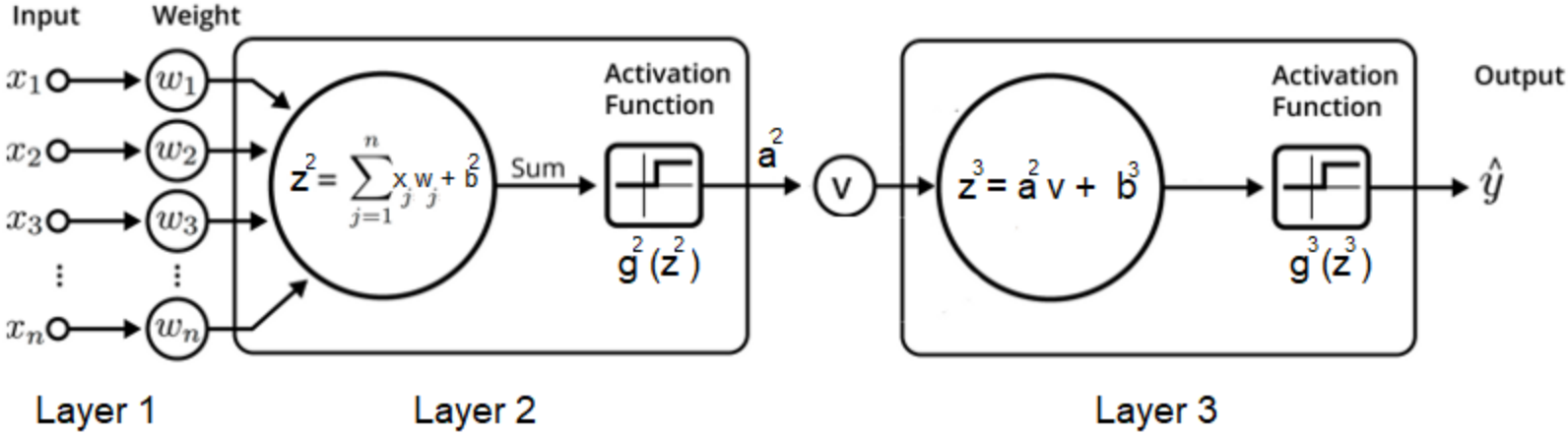
The same logic will be applied in the case of the neural network. For each i, $w_i^+ = w_i - \alpha\partial\mathcal{L}/\partial w_i = w_i - \alpha\delta * x_i$ ***where*** w_i^+ is the new value of w_i. Impo**rtant: for** each i we iterate w_i e**nou**gh number of time until $\partial\mathcal{L}/\partial \mathbf{w_i}$ **becom**es small enough

Using the same logic we can also find the effect of the variation of ***b***: $\partial\mathcal{L}/\partial b$ __ $=\partial\mathcal{L}/\partial a\; \partial a/\partial z\; \partial z/\partial b$_S_ince $\partial z/\partial b = 1$ and all the other terms have been already computed _∂_$\mathcal{L}/\partial b = \delta$ *it f*ollows that b^+ **= b – __ $\alpha\partial\mathcal{L}/\partial b = b - \alpha\delta$ whe_re b^+ is th_e new value of b.

Aga_i_n we keep iterating b enou_g_h **number of time until ∂ℒ/∂b b**ecom_e_s** small enough.

Case of Two Neurons

Now let's take the case of two neurons put in sequence (they form two layers).



Before starting it is worth noting that the notation has changed to accommodate the new architecture. The superscript number denotes the layer. For example z^2, b^2, a^2, g^2 belong to layer #2, while z^3, b^3, a^3, g^3 belong to layer #3. It is also worth mentioning that the output of layer #2 is a single value $\mathbf{a^2}$, which is multiplied by the weight **v** in the next layer.

Layer #3

We will start with the last layer (layer #3) first, because its result will be used in the previous layer.

The chain of events is nothing unusual. Variation of **v** causes $\mathbf{z^3}$ to vary, variation of $\mathbf{z^3}$ causes $\mathbf{g^3(z^3)}$ to vary, variation of $\mathbf{g^3(z^3)}$ causes $\mathbf{\mathcal{L}(y, \hat{y})}$ to vary.

Using the chain rule we compute the derivative $\partial\mathcal{L}/\partial v$: $\boldsymbol{\partial\mathcal{L}/\partial v = \partial\mathcal{L}/\partial a^3\ \partial a^3/\partial z^3\ \partial z^3/\partial v}$

As previously done we compute each term alone:

$\boldsymbol{\partial\mathcal{L}/\partial a^3 = a^3 - y}$

$\boldsymbol{\partial a^3/\partial z^3 = g^{3\prime}(z^3)}$ (where $g^{3\prime}(z^3)$ is the derivative of $g^3(z^3)$)

$\boldsymbol{\partial z^3/\partial v = a^2}$

So $\partial\mathcal{L}/\partial v$ becomes $\boldsymbol{\partial\mathcal{L}/\partial v = (a^3 - y)\ g^{3\prime}(z^3)\ a^2}$

Let's define $\boldsymbol{\delta^3 = (a^3 - y)\ g^{3\prime}(z^3)}$ ***It follows that*** $\boldsymbol{\partial\mathcal{L}/\partial v = \delta^3\ a^2}$

Similarly we can find $\boldsymbol{\partial\mathcal{L}/\partial b^3 = \delta^3}$

Layer #2

Now let's move to layer #2. The chain reaction starts by applying variation to $\mathbf{w_i}$ which affects affects $\mathbf{z^2}$, variation of $\mathbf{z^2}$ affects $\mathbf{g^2(z^2)}$ variation of $\mathbf{g^2(z^2)}$ affects $\mathbf{z^3}$ (note that at this point **v** is considered fixed) variation of $\mathbf{z^3}$ affects $\mathbf{g^3(z^3)}$ variation of $\mathbf{g^3(z^3)}$ affects $\mathbf{\mathcal{L}(y, \hat{y})}$

The derivative equation is $\boldsymbol{\partial\mathcal{L}/\partial w_i = (\partial\mathcal{L}/\partial a^3\ \partial a^3/\partial z^3\ \partial z^3/\partial a^2)\ \partial a^2/\partial z^2\ \partial z^2/\partial w_i}$

We have already calculated $\boldsymbol{\partial\mathcal{L}/\partial a^3}$ and $\partial a^3/\partial z^3$, as for the rest they can be easily derived.

$\partial z^3/\partial a^2 = \partial(a^2 * v))/\partial a^2 = v^{**}\partial a^2/\partial z^2 = \boldsymbol{\partial g^2(z^2)/\partial z^2 = g^{2\prime}(z^2)}$ $\partial z^2/\partial w_i = x_i^{**}$

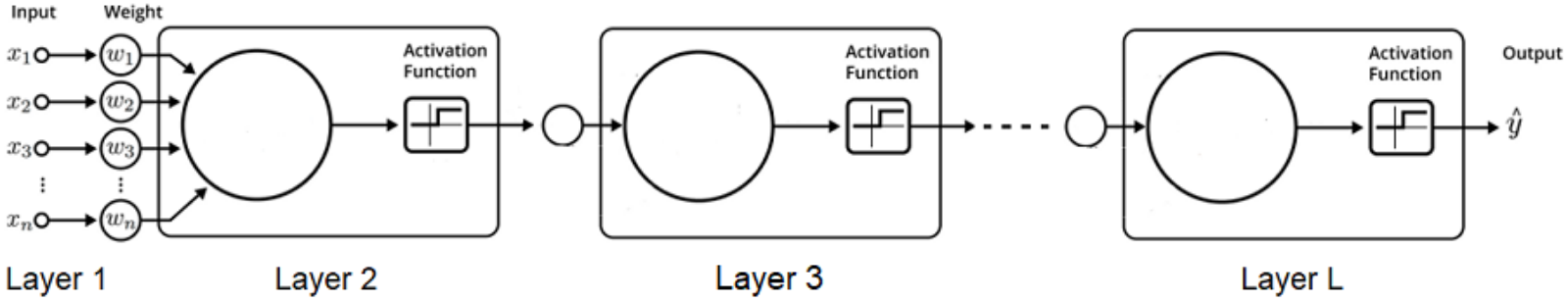
After replacing each term but its value $\partial\mathcal{L}/\partial w_i = ((a^3 * \boldsymbol{- y)\ g^{3\prime}(z^3)\ v)\ g^{2\prime}(z^2)\ x_i}$ ***We already defined*** $\boldsymbol{\delta^3}$ ***as*** $\boldsymbol{\delta^3 = (a^3 - y)\ g^{3\prime}(z^3)}$ $\boldsymbol{\partial\mathcal{L}^*/\partial w_i = \delta^3\ v\ g^{2\prime}(z^2)\ x_i}$

Lets define $\boldsymbol{\delta^2}$ as $\boldsymbol{\delta^2 = \delta^3\ v\ g^{2\prime}(z^2)}$ **Th**is will give the final form of $\partial\mathcal{L}/\partial w_i$: $\partial\mathcal{L}/^*\partial w_i = \delta^2\ x_i^{**}$

Similarly we can find $\partial\mathcal{L}/\partial b^2 = \delta^2$

Sequence of L Layers

Now consider a network with L layers each layer containing only one neuron. Such network will look like the image below.



By using the same logic as above we can find the different components needed for the Back Propagation.

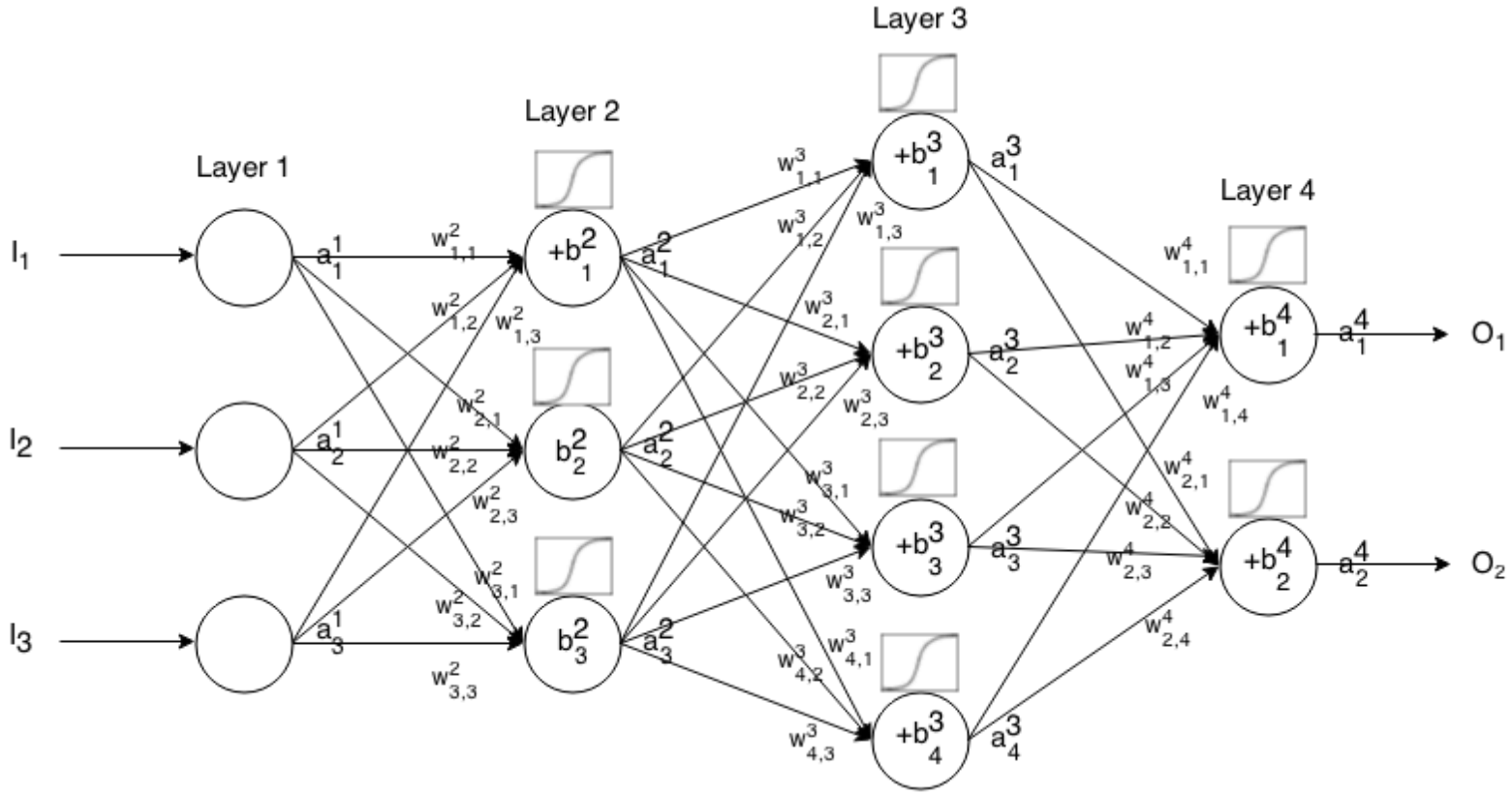
So for layer L $\delta^L =$ (**$a^L - y$) $g'^L(z^L)$ ***For any other layer $l < L$*** $\delta^l = \delta^{l+1} w^{l+1} * g'(z^l)$**

For any layer $l \leq L$ $*\partial\mathcal{L}/\partial w^l = \delta^l a^{l-1}$ $\partial\mathcal{L}/\partial b^l = \delta^l$ where **a^{l-1} is**** the output of the layer l-1, or if we are at layer 1 it will be the input x.

The formula for the gradient descent to find the **w_i^l** (weight at layer l, neuron i) that $w_i^{k} =$ **$w_i^l - \alpha\partial\mathcal{L}/\partial w_i^l = w_i^l - \alpha\delta^{l} * a^{l-1}$ *where*** w_i^{l+} i_**s the new value of w_i^l. I**_m_portant: for e**ach w_i^l we __ iterate enough number of times until $\partial\mathcal{L}/\partial w_i^l$** be_c_ome**s small enough

General Case

Multiple Layers with Multiple Neurons



In this section we have multilayer neural network with multi neurons in each layer instead of one as we have used so far. Recall that in the previous cases we had one neuron per layer and we got the $\delta^L =$ (**$a^L - y$) $g'^L(z^L)$ ***for layer L and*** $\delta^l = \delta^{l+1} w^{l+1} * g'(z^l)$ for **layer $l <$ L This is only for one neuron in each layer. When we have multiple neurons per**

layer, we have to compute δ for each neuron of each layer. So now variable with only superscript letter such as δ^L, $\mathbf{w}^l$ are vectors and matrices, whereas variables with both superscript and subscript letters such as δ^l_i and w^l_{ir} are single values. Since we have multiple output layer, the definition of loss function $\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$ is no more sufficient. We have to account to all output neurons. So we define the cost function as the square sum of all output neurons $C=\frac{1}{2}\sum_i (y_i - \hat{y}_i)^2$. ____**
So the formulas that we have computed so far will have a vector form:

$$\delta^L = \nabla_a C \odot \mathbf{g}^{L\prime}(z^L)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \mathbf{g}^{l\prime}(z^l)$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

Where $\nabla a C$ ** **is a vector of variation of the cost** C **relative to the output of the network** a^L_i**, which is** $\partial C / \partial a^L_i$.** The $\odot$ operator is a member-wise vector/matrix multiplication.

The Back Propagation Algorithm

1. **Input x:** Set the input in layer 1.
2. **Forward:** For each layer l = 2,3,…,L we compute $\mathbf{z}^l = \mathbf{w}^l \mathbf{a}^{l-1} + \mathbf{b}^l$ and $\mathbf{a}^l = \mathbf{g}^l(\mathbf{z}^l)$.
3. **Output error δ^L:** At the output layer we compute the vector $\delta^L = \nabla a C \odot g^{L\prime}(z^L)$**.** This will be the start of the back propagation.
4. **Back propagation:** We move backward, for each layer l=L-1,L-2,L-3,…,2 we compute the error of each layer $\delta^l = ((w^{l+1})^T * \delta^{l+1}) \odot g''(z^l)$**. Then we update the weights of each layer using the Gradient Descent Formula:** $w^{l+}_{i-r} = w^l_{ir} - \alpha * \delta^l_i * a^{l-1}_r$ ***and*** $b^{l+}_i = b^l_i - \alpha * \delta^l_i$**where** w^{l+} **and** b^{l+} **are** the updated values of w^l **and** b^l **at l**ayer l af**te**r each iteration.
5. **Output:** At the end we will have the weights **w** and biases **b** at each layer that have been computed to minimize the cost function **C**.

Back Propagation, the Easy Way (Part 2)

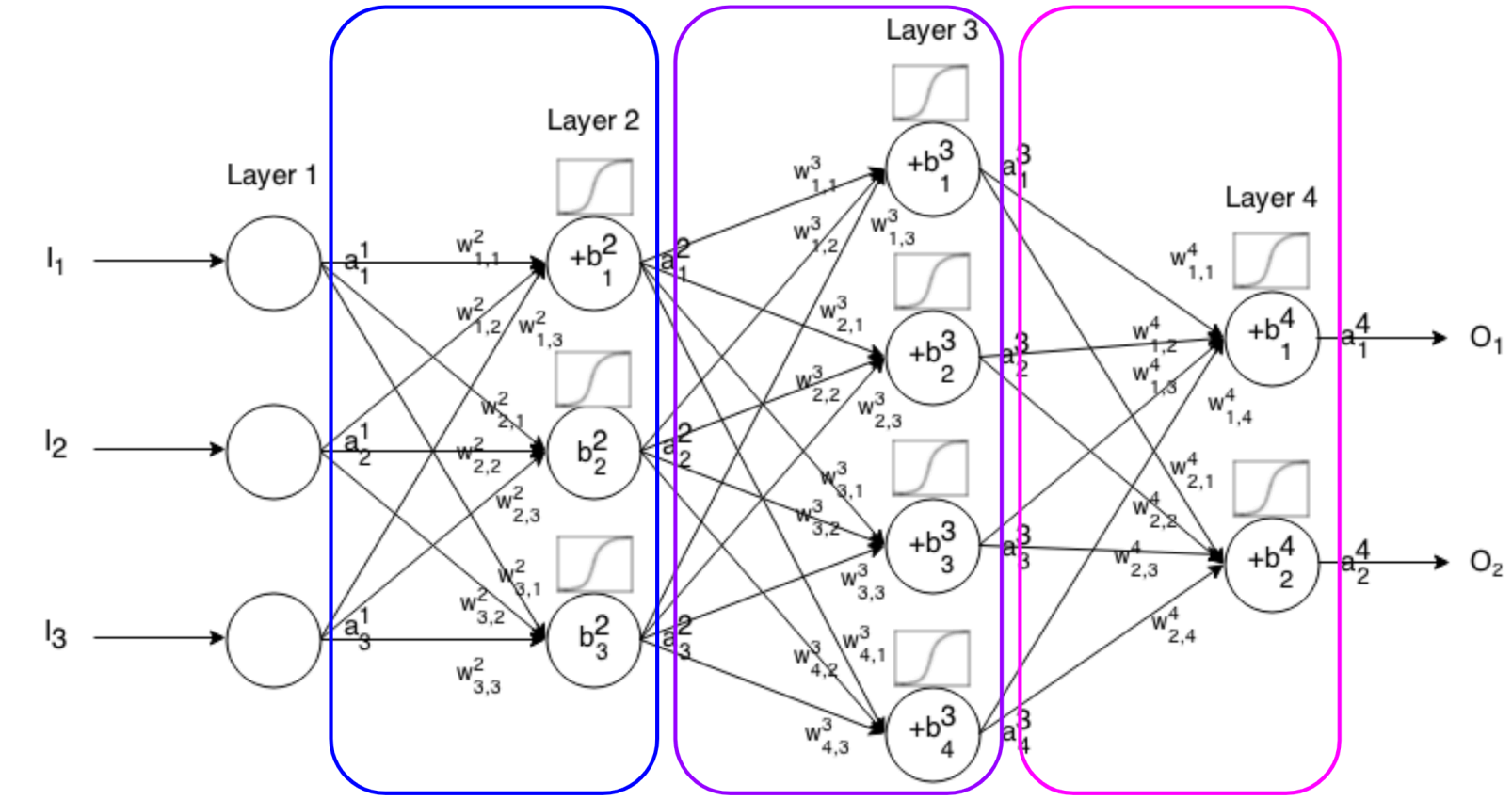
Practical implementation

Ziad SALLOUM
Jan 19, 20196 min read
of Back Propagation

Update: The best way of learning and practicing Reinforcement Learning is by going to http://rl-lab.com
In the first part we have seen how back propagation is derived in a way to minimize the cost function. In this article we will see the implementation aspect, and some best practices to avoid common pitfalls.
We are still in the simple mode, where input is handled one at a time.

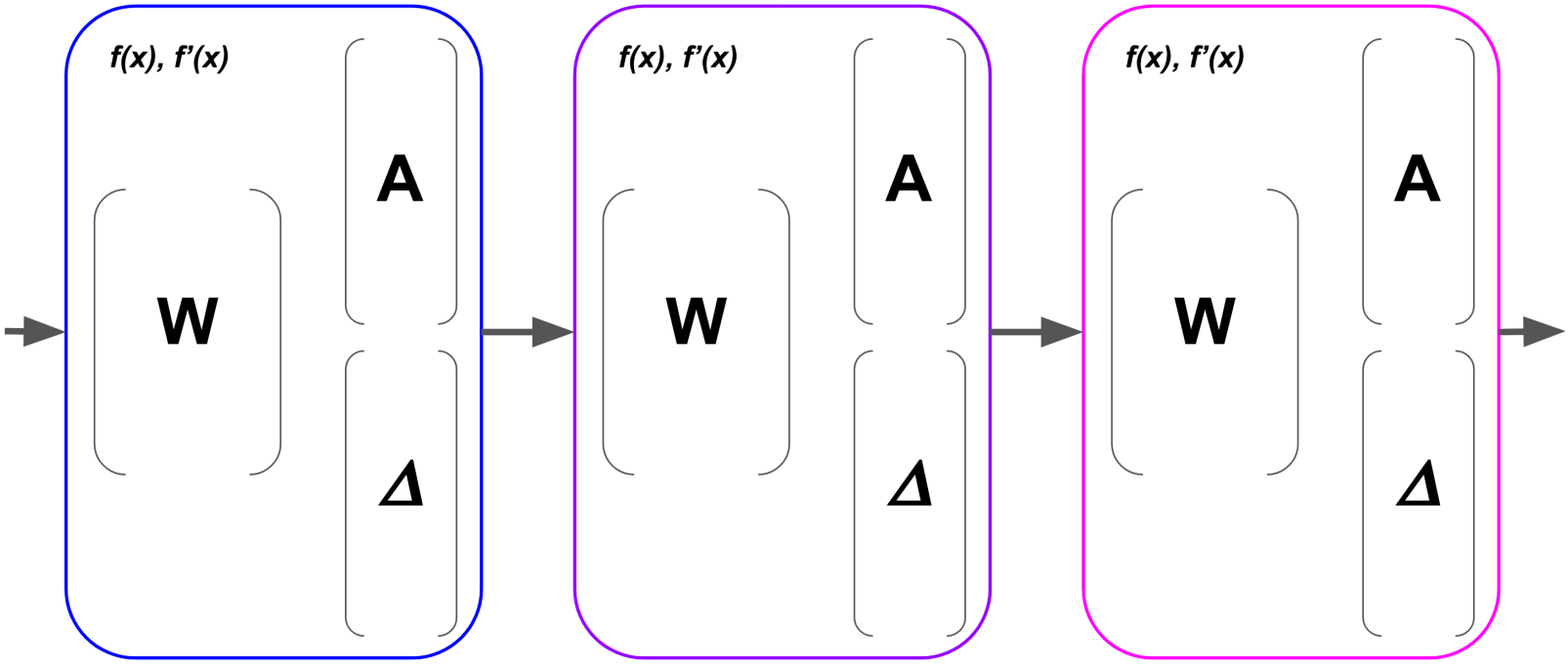
Layer Class

Consider a fully connected neural network such as in the figure below.



Each layer will be modelled by a Layer object containing the weights, the activation values (output of the layer), the gradient dZ (not represented in the image), the cumulative error delta (Δ), as well as the activation function f***(x)*** and its derivative f'***(x).*** The reason for storing intermediate is to avoid computing them each time they are needed.

Advice: It is better to organize the code around few classes, and avoid cramming everything into arrays, as it is very easy to get lost.



Note that the input layer won't be represented by a Layer object since it consists only of a vector.

```
class Layer:

    def __init__(self, dim, id, act, act_prime,
                 isoutputLayer = False):
        self.weight = 2 * np.random.random(dim) - 1
```

```
        self.delta = None
        self.A = None
        self.activation = act
        self.activation_prime = act_prime
        self.isoutputLayer = isoutputLayer
        self.id = id
```

The constructor of the Layer class, takes as parameters:

- dim: dimensions of the weight matrix,
- id: integer as id of the layer,
- act, act_prime: the activation function and its derivative,
- isoutputlayer: True if this layer is the output, False otherwise.

It initializes the weights randomly to numbers between -1 and +1, and set the different variables to be used inside the object.

The layer object has three methods:

- forward, to compute the layer output.
- backward, to propagate the error between the target and the output back to the newtwork.
- update, to update the weights according to a Gradient Descent.

```
def forward(self, x):
    z = np.dot(x, self.weight)
    self.A = self.activation(z)
    self.dZ = self.activation_prime(z);
```

The forward function, computes and returns the output of the Layer, by taking the input **x** and computes and stores the output A = activation (W.X). It also computes and stores dZ which the derivative of the output relative to the input.

The backward functions takes two parameters, the target y and rightLayer which is the layer (ℓ-1) assuming that the current one is ℓ.

It computes the cumulative error delta that is propagating from the output going leftward to the beginning of the network.

IMPORTANT: a common mistake, is to think that the backward propagation is some kind of loopback in which the output is injected again in the network. So instead of using _**dZ = self.activation_prime(z);** *some uses* **self.activation_prime(A)**._ This is wrong, simply because what we are trying to do is figure out how the output A would vary relative to input z. This means computing the derivative ***∂a/∂z*** = ∂g(z)/∂z = **g'(z)** according to the chain rule.****This error might be due to the fact that in the case of sigmoid activation function a** = *σ*(z), **the derivative *σ*'**(z) = *σ*(z)*(1-σ(z)) = a*(1-a). Which **gives the illusion that the output is injected into to the network, while the truth is that we are computing *σ*'(z)**.**

```
def backward(self, y, rightLayer):
    if self.isoutputLayer:
        error =  self.A - y
        self.delta = np.atleast_2d(error * self.dZ)
    else:
        self.delta = np.atleast_2d(
            rightLayer.delta.dot(rightLayer.weight.T)
            * self.dZ)
    return self.delta
```

What the backward function does is to compute and return the delta, based on the formula:

$$\delta^L = \nabla_a C \odot \mathbf{g}^{L\prime}(z^L)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \mathbf{g}^{l\prime}(z^l)$$

Finally the update function uses the gradient descent to update the weights of the current layer.

```
def update(self, learning_rate, left_a):
    a = np.atleast_2d(left_a)
    d = np.atleast_2d(self.delta)
    ad = a.T.dot(d)
    self.weight -= learning_rate * ad
```

NeuralNetwork class

As one might guess layers form a network, so the class NeuralNetwork is used to organize and coordinate the layers. It's constructor takes the configuration of the layers that is an array which length determines the number of layers in the network and each element defines the number of nodes in the corresponding layer. For example [2, 4, 5,] means that the network has 4 layers with the input layer having 2 nodes, the next hidden layers have 4 and 5 nodes respectively and the output layer has 1 node. The second parameter is the type of activation function to use for all layers.

The fit function is where all the training happens. It starts by selecting one input sample, computes the forward over all the layers, then computes the error between the output of the network and the target value and propagate this error to the network by calling backward function of each layer in reverse order, starting by the last one up to the first. Finally, the update function is called for each layer to update the weights.

These steps are repeated a number of times determined by the parameter epoch.

After the training is complete, the predict function can be called to test input. The predict function is simply a feed forward of all the network.

```
class NeuralNetwork:

    def __init__(self, layersDim, activation='tanh'):
        if activation == 'sigmoid':
            self.activation = sigmoid
            self.activation_prime = sigmoid_prime
        elif activation == 'tanh':
            self.activation = tanh
            self.activation_prime = tanh_prime
        elif activation == 'relu':
            self.activation = relu
            self.activation_prime = relu_prime

        self.layers = []
        for i in range(1, len(layersDim) - 1):
            dim = (layersDim[i - 1] + 1, layersDim[i] + 1)
            self.layers.append(Layer(dim, i, self.activation, self.activation_prime))

        dim = (layersDim[i] + 1, layersDim[i + 1])
        self.layers.append(Layer(dim, len(layersDim) - 1, self.activation, self.activation_prime, True
# train the network
    def fit(self, X, y, learning_rate=0.1, epochs=10000):
        # Add column of ones to X
        # This is to add the bias unit to the input layer
        ones = np.atleast_2d(np.ones(X.shape[0]))
        X = np.concatenate((ones.T, X), axis=1)

        for k in range(epochs):

            i = np.random.randint(X.shape[0])
            a = X[i]

            # compute the feed forward
            for l in range(len(self.layers)):
                a = self.layers[l].forward(a)

            # compute the backward propagation
            delta = self.layers[-1].backward(y[i], None)

            for l in range(len(self.layers) - 2, -1, -1):
                delta = self.layers[l].backward(delta, self.layers[l+1])

            # update weights
            a = X[i]
            for layer in self.layers:
                layer.update(learning_rate, a)
                a = layer.A
# predict input
    def predict(self, x):
        a = np.concatenate((np.ones(1).T, np.array(x)), axis=0)
        for l in range(0, len(self.layers)):
            a = self.layers[l].forward(a)
        return a
```

Running The Network

To run the network we take as example the approximation of the Xor function.

A	B	A **XOR** B
0	0	0
0	1	1
1	0	1
1	1	0

We try the several network configurations, using different learning rates and epoch iterations. Results are liste below:

```
Result with tanh
[0 0] [-0.00011187]
[0 1] [ 0.98090146]
[1 0] [ 0.97569382]
[1 1] [ 0.00128179]
 Result with sigmoid
[0 0] [ 0.01958287]
[0 1] [ 0.96476513]
[1 0] [ 0.97699611]
[1 1] [ 0.05132127]
 Result with relu
[0 0] [ 0.]
[0 1] [ 1.]
[1 0] [ 1.]
[1 1] [ 4.23272528e-16]
```

It is advisable that you try different configurations and see for yourself which one gives the best and most stable results.

Back Propagation, the Easy Way (Part 3)

How to handle dimensions of matrices

Ziad SALLOUM
Jan 20, 20195 min read

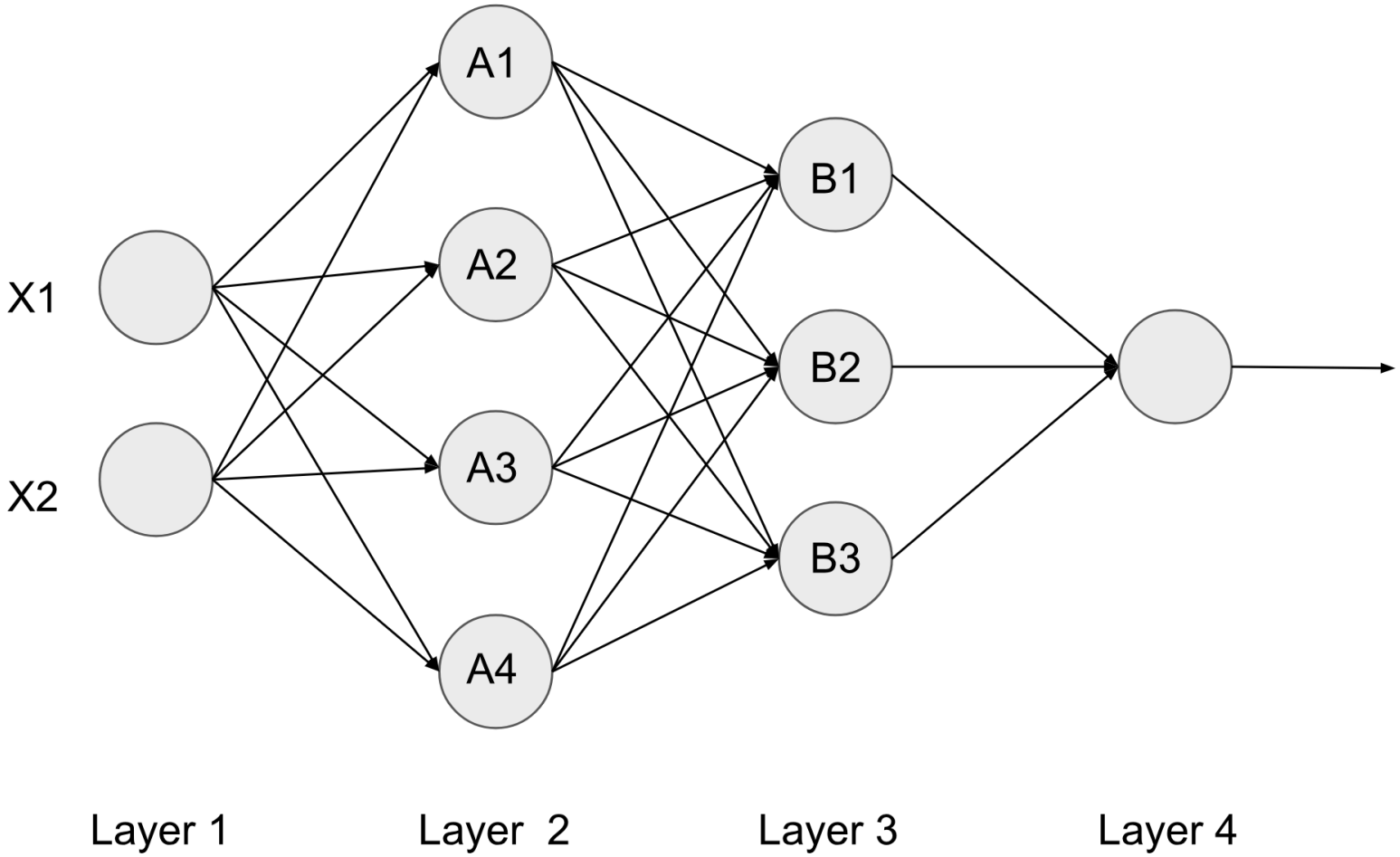
Photo by Todd Brogowski on Unsplash

Update: The best way of learning and practicing Reinforcement Learning is by going to http://rl-lab.com

Up until know, in part 2 of the series, we have considered that our neural network handles input samples, one at a time. However this is not efficient. This takes too much computing and does not benefit from lots of optimization techniques used in the matrix computations.

Most of the time the data used to train a neural network comes in the format of files full of rows, where each row represent a sample and each column represent a feature.

Let's consider the following network.



Feed Forward

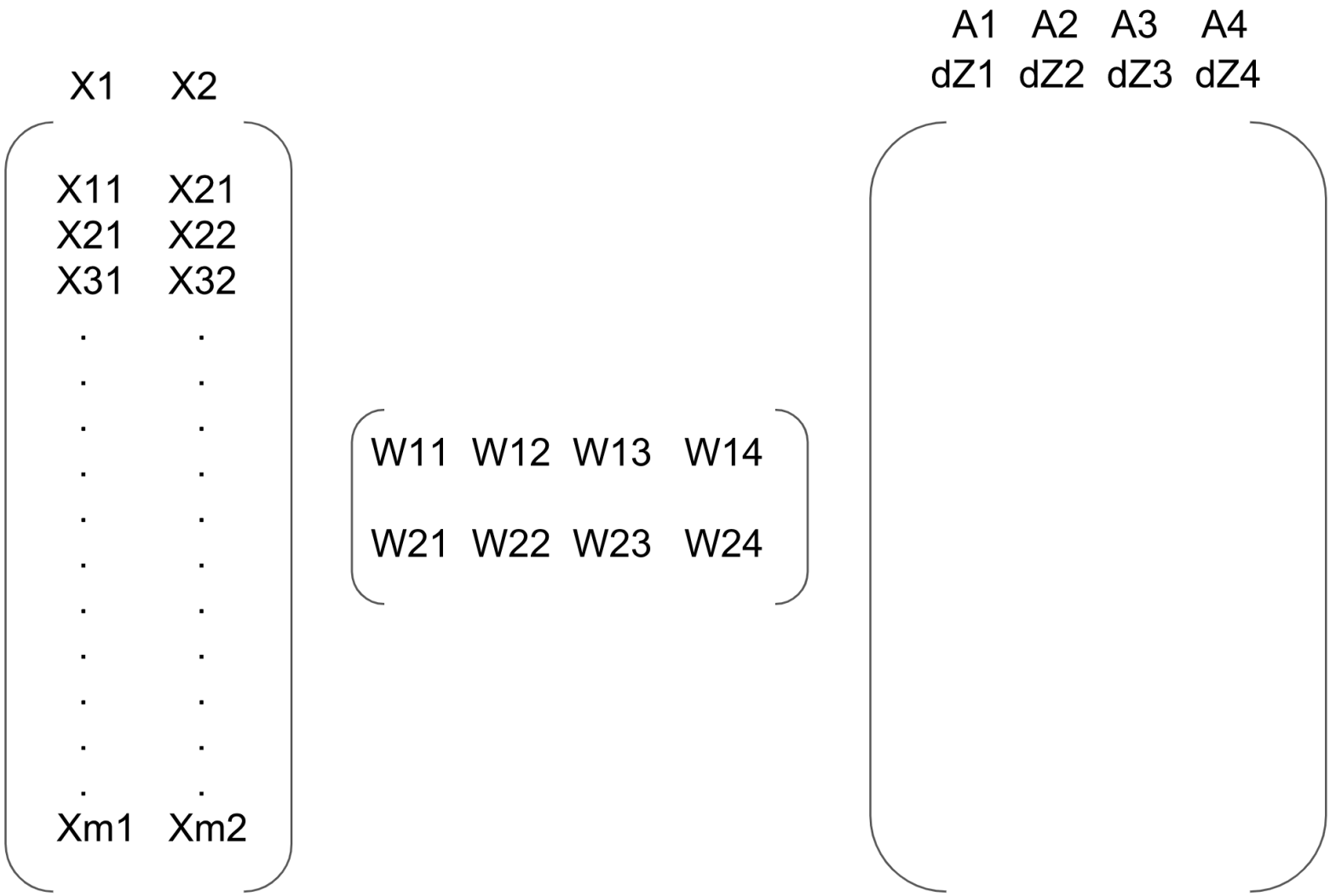
We will start by focusing on the first two layers.

The input is a file containing rows of data, in which each column contains the values to go into one input entry of the neural network. For example the column X1 with values (X11, X21, …, Xm1) goes to the X1 input of the neural network, similarly to the column X2.

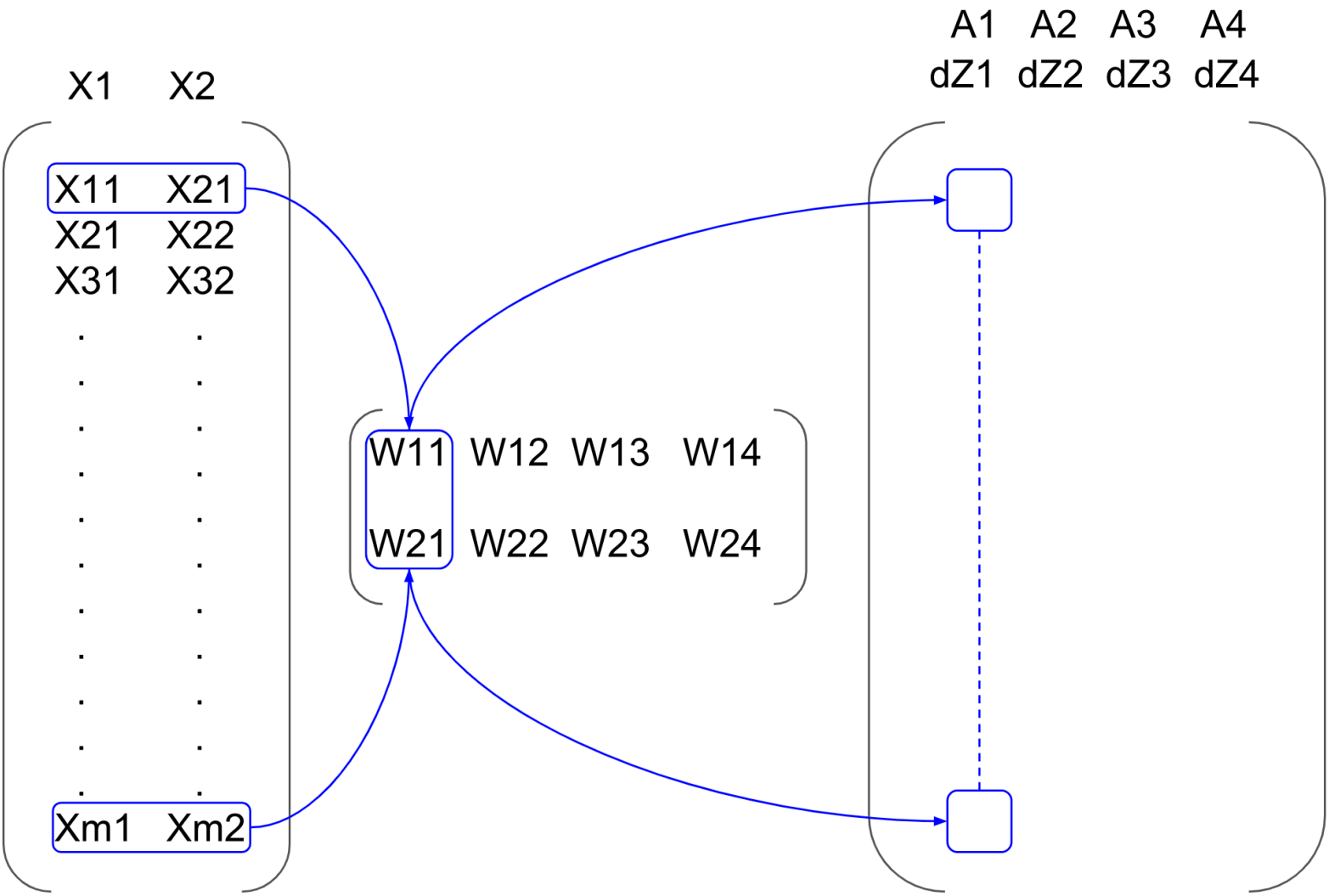
It is natural to think that the number of rows that is fed into the neural network will be collected at the output. Meaning if we have two samples that are fed into the network one at a time, we should expect to have two results (one at a time) at the output. Same thing if we fed the data as matrix containing two rows representing the two samples, we should expect to have a matrix having two rows of results.

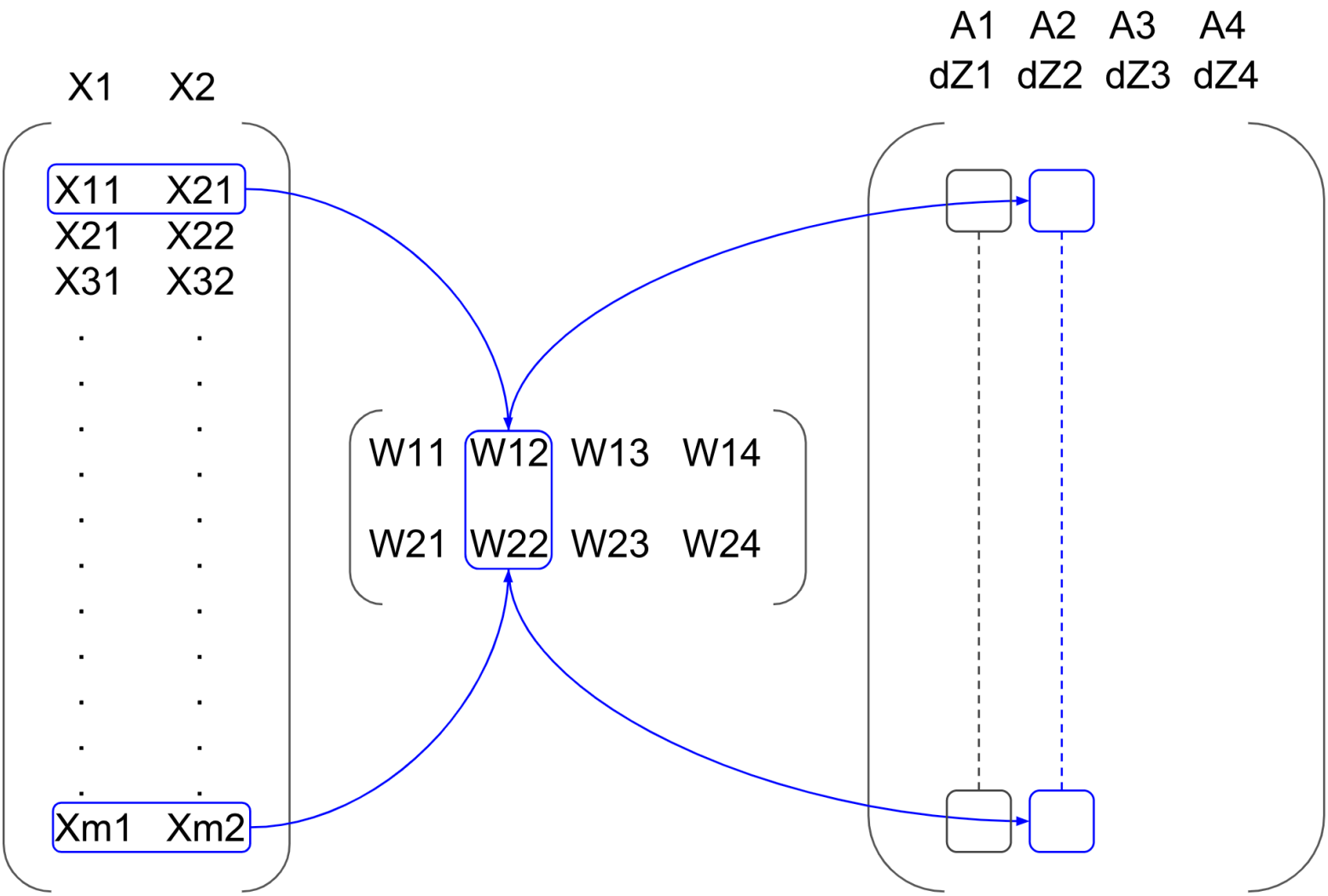
This applies to the output of each layer as well. Let's consider a file containing the different samples for X1 and X2. At the output of Layer 2, we have for values A1, A2, A3, A4 for each sample that is fed to the input layer. This is true for each sample, so for **m** samples we will have **m** outputs such as (A11, A12, A13, A14), (A21, A22, A23, A24),….(Am1, Am2, Am3, Am4). So the dimension of the A matrix at the output of Layer 2 is (m, 4). Same goes for the derivative of Z.

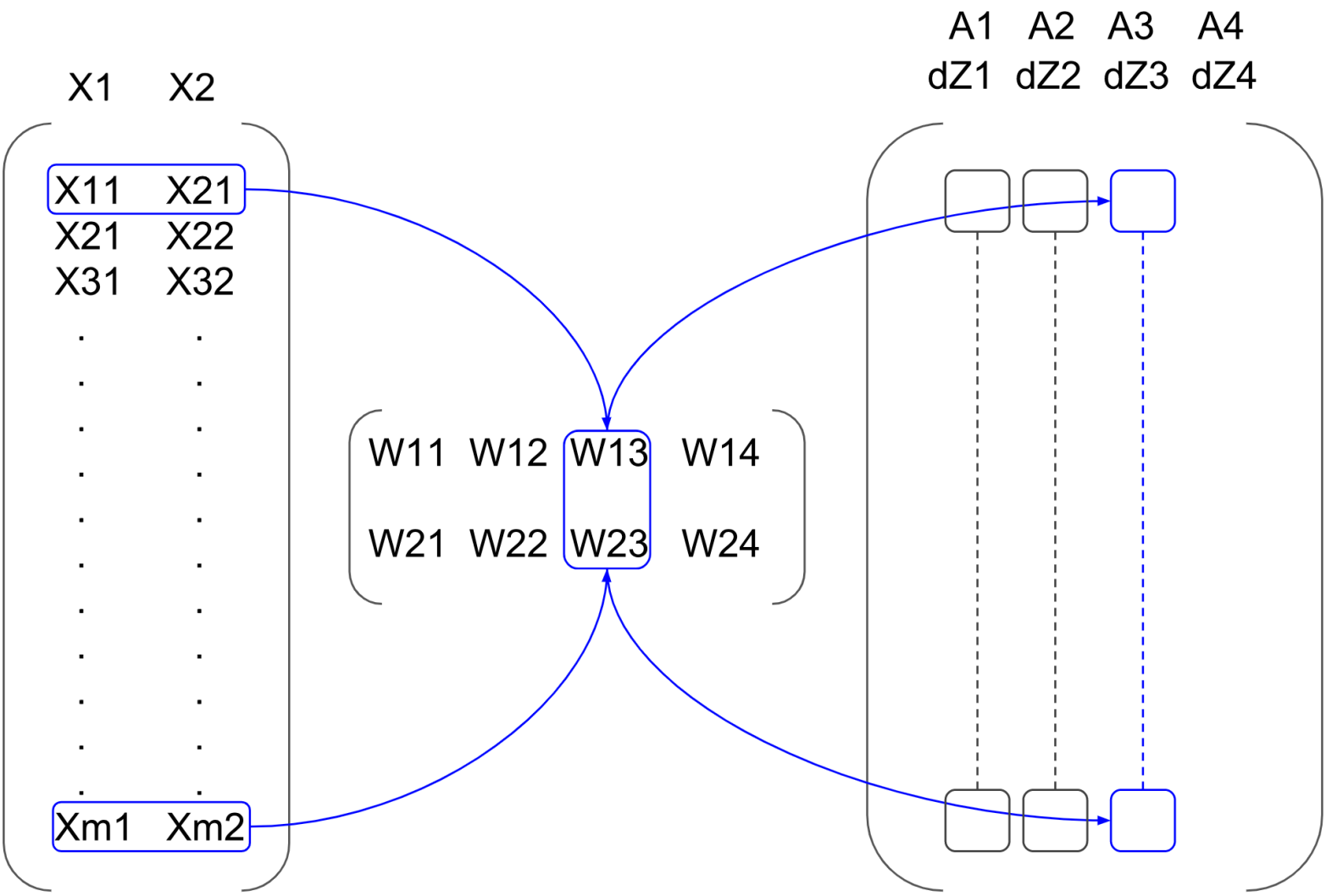
Note that the dimension of the weight matrix does not change with the number of samples, it only depends of the number of nodes at the input (2 in the example) and the nodes at the output (4 in the example), which makes it (2, 4).

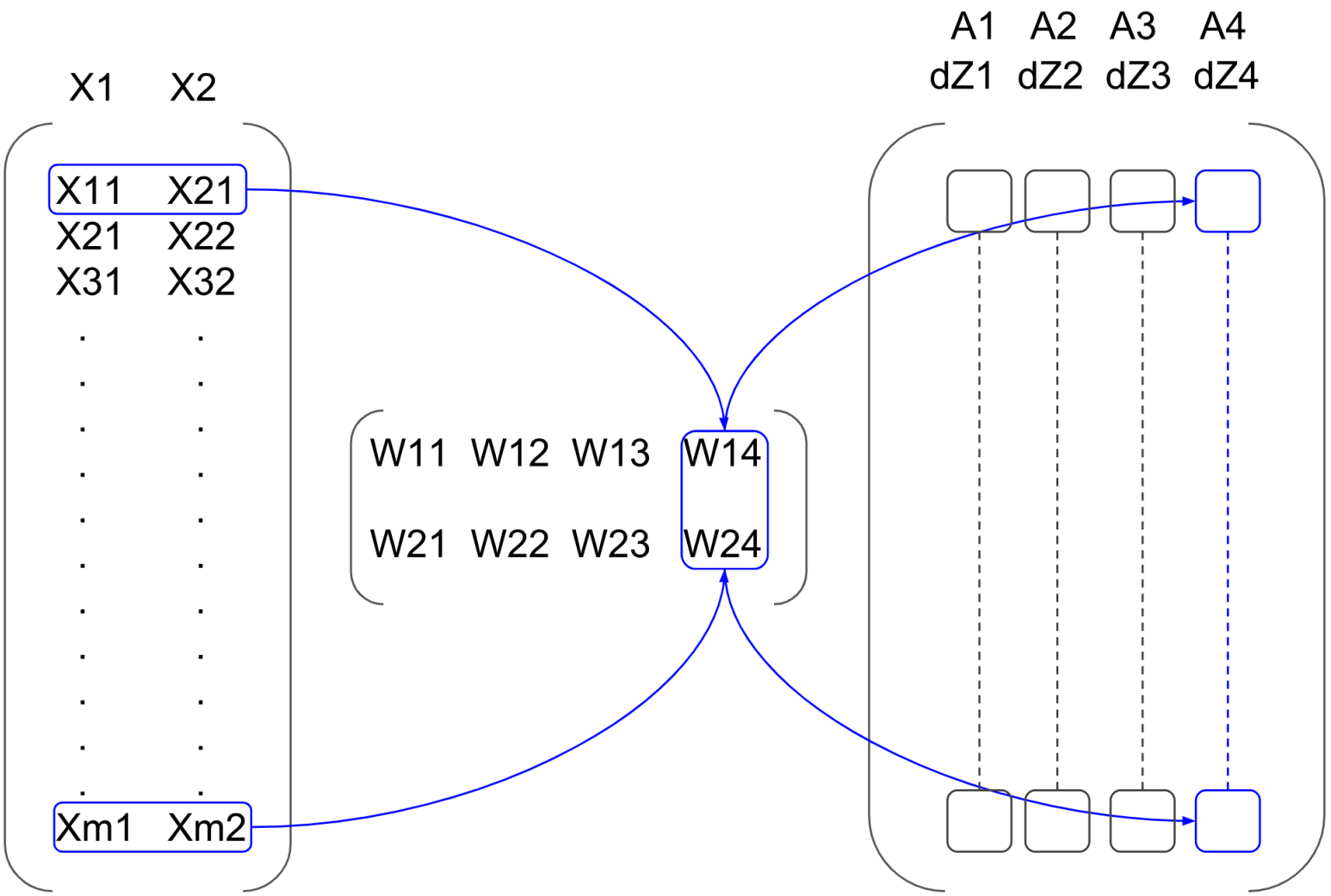


Alternatively we can obtain the same result via linear algebra. As we already know that in the feed froward we compute A and dZ such as **A = f(X.W)** and **dZ = f'(X.W)**. So to compute the Dimensions of A and dZ, it suffices to look at the dimensions of X and W, which are (m, 2) and (2, 4) respectively. The result is (m, 2) x (2, 4) = (m, 4). The following pictures depicts how the A and dZ are filled to form matrices of dimension (m, 4).


A1 A2 A3 A4
dZ1 dZ2 dZ3 dZ4
X1 X2
X11 X21
X21 X22
X31 X32
Xm1 Xm2
W11 W12 W13 W14
W21 W22 W23 W24


A1 A2 A3 A4
dZ1 dZ2 dZ3 dZ4
X1 X2
X11 X21
X21 X22
X31 X32
Xm1 Xm2
W11 W12 W13 W14
W21 W22 W23 W24


A1 A2 A3 A4
dZ1 dZ2 dZ3 dZ4
X1 X2
X11 X21
X21 X22
X31 X32
Xm1 Xm2
W11 W12 W13 W14
W21 W22 W23 W24



Back Propagation

So far we computed the dimensions of the output of every layer in the Feed Forward phase. Let's look at the Back Propagation.

As a reminder, the formulas in the Back Propagation are:

$\Delta^{L} = (A^{L} - Y) * dZ^{L}$

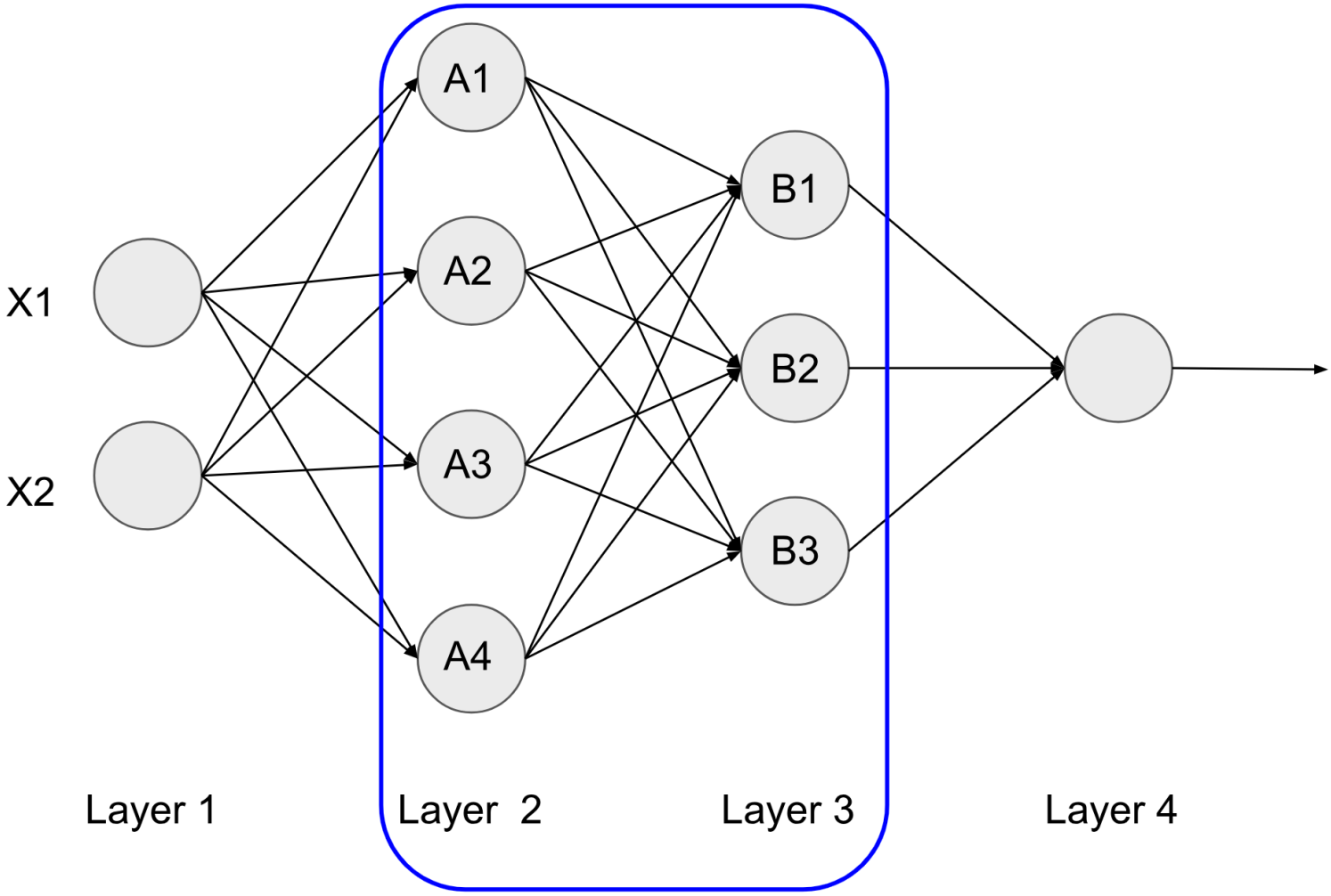
$\Delta^{i} = (\Delta^{i+1} . W^{T}) * dZ^{i}$

(where * is a member wise multiplication)

We know by now that both A and dZ have the dimensions (m, n) where **m** is the number of samples and **n** is the number of nodes in the layer. So the Δ^{L} will have the same dimension (m, n)

In our example the last layer (L) has only one node, so A and dZ at layer L have the dimension (m, 1). It follows that Δ^{L} has also the dimension (m, 1).

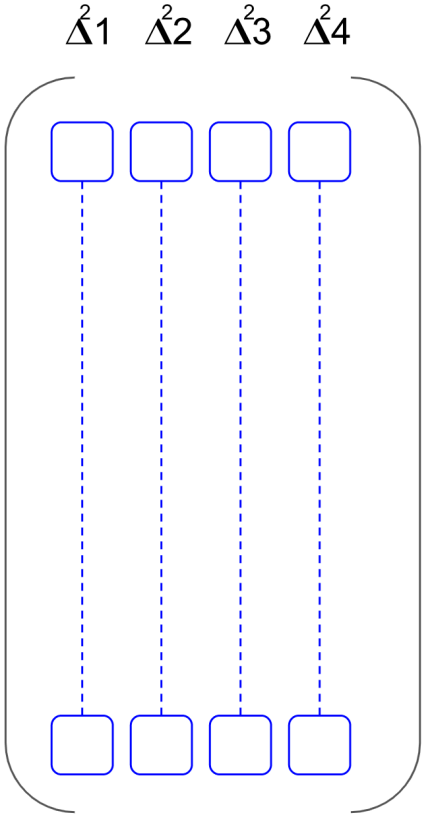
In order to compute the deltas of the inner layers, let's take layers 2 and 3 of the example.

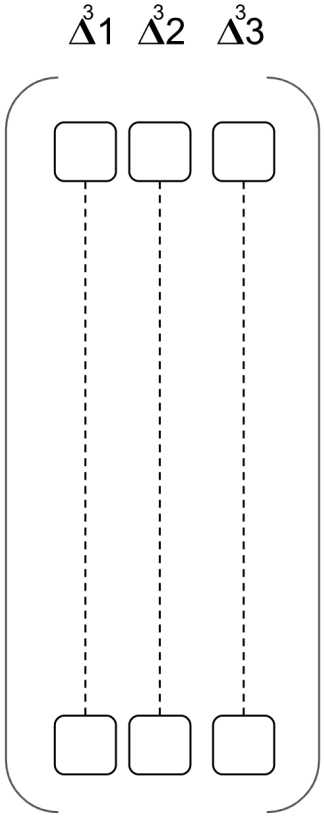


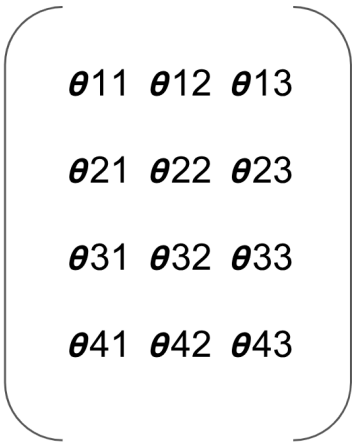
The weight θ between layer 2 and layer 3 has the dimension (4, 3), since the 4 nodes of layer 2 are connected to 3 nodes of layers 3.

$$\begin{pmatrix} \theta 11 & \theta 12 & \theta 13 \\ \theta 21 & \theta 22 & \theta 23 \\ \theta 31 & \theta 32 & \theta 33 \\ \theta 41 & \theta 42 & \theta 43 \end{pmatrix}$$

However since the delta at layer 2 is the dot product of the delta coming from layer 3 and the weight θ, we end up with the following configuration.



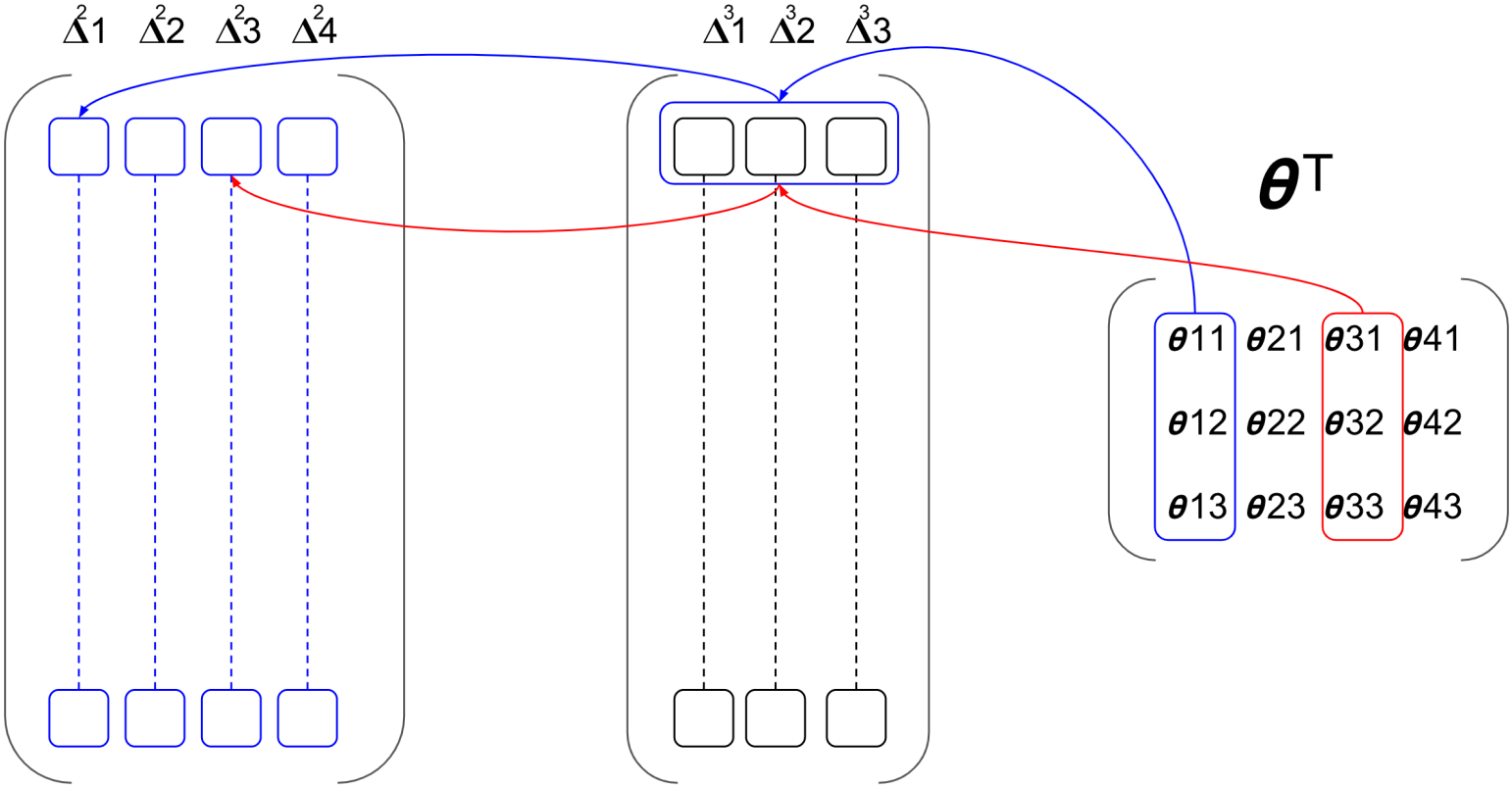


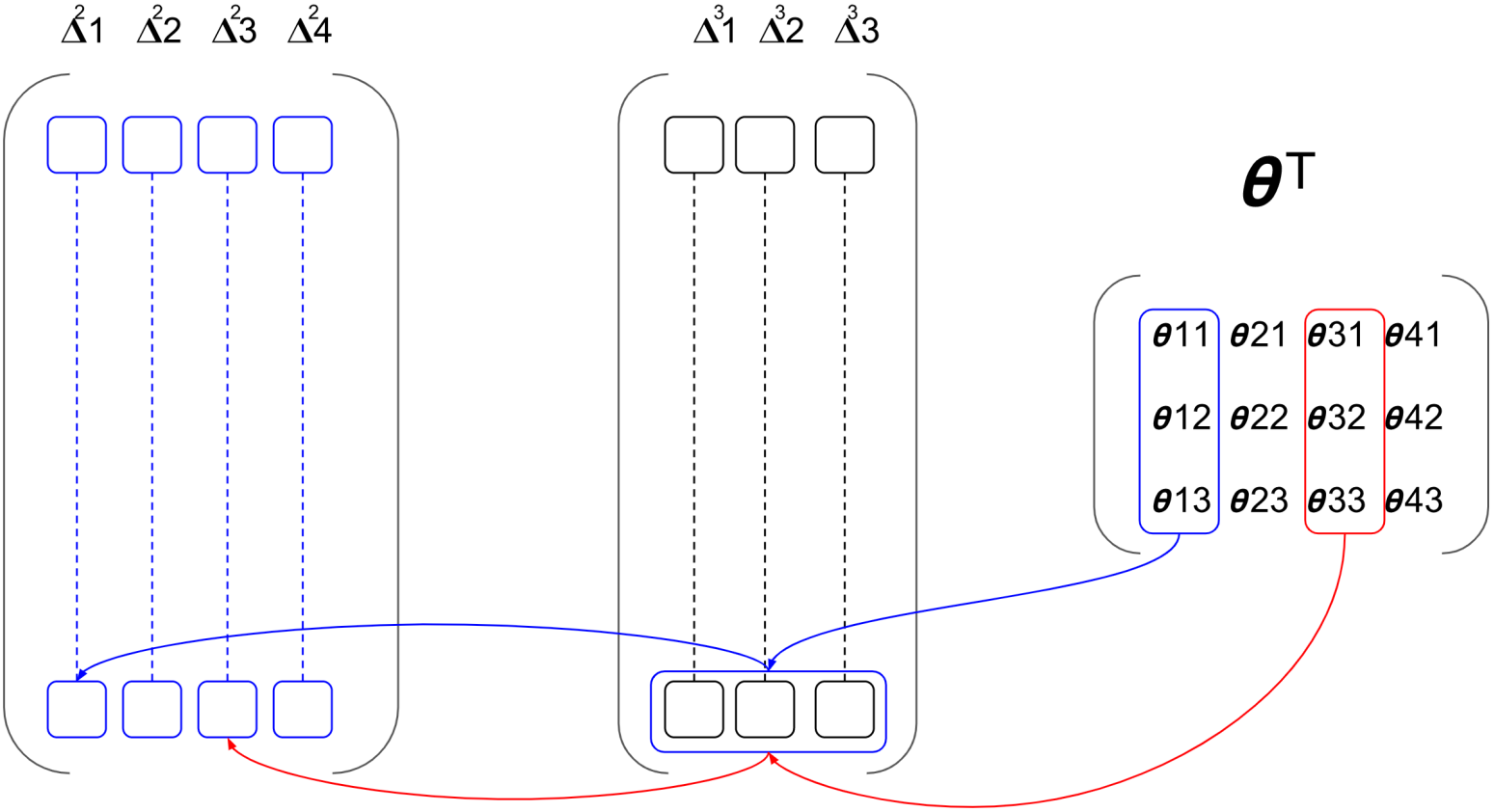


?

[m, 4] [m, 3] [4, 3]

It not possible to do the dot product and obtain delta at layer 2 with dimension (m, 4)! To solve this, we use θ^T the transpose of θ.
The computation will be possible as shown by the images below

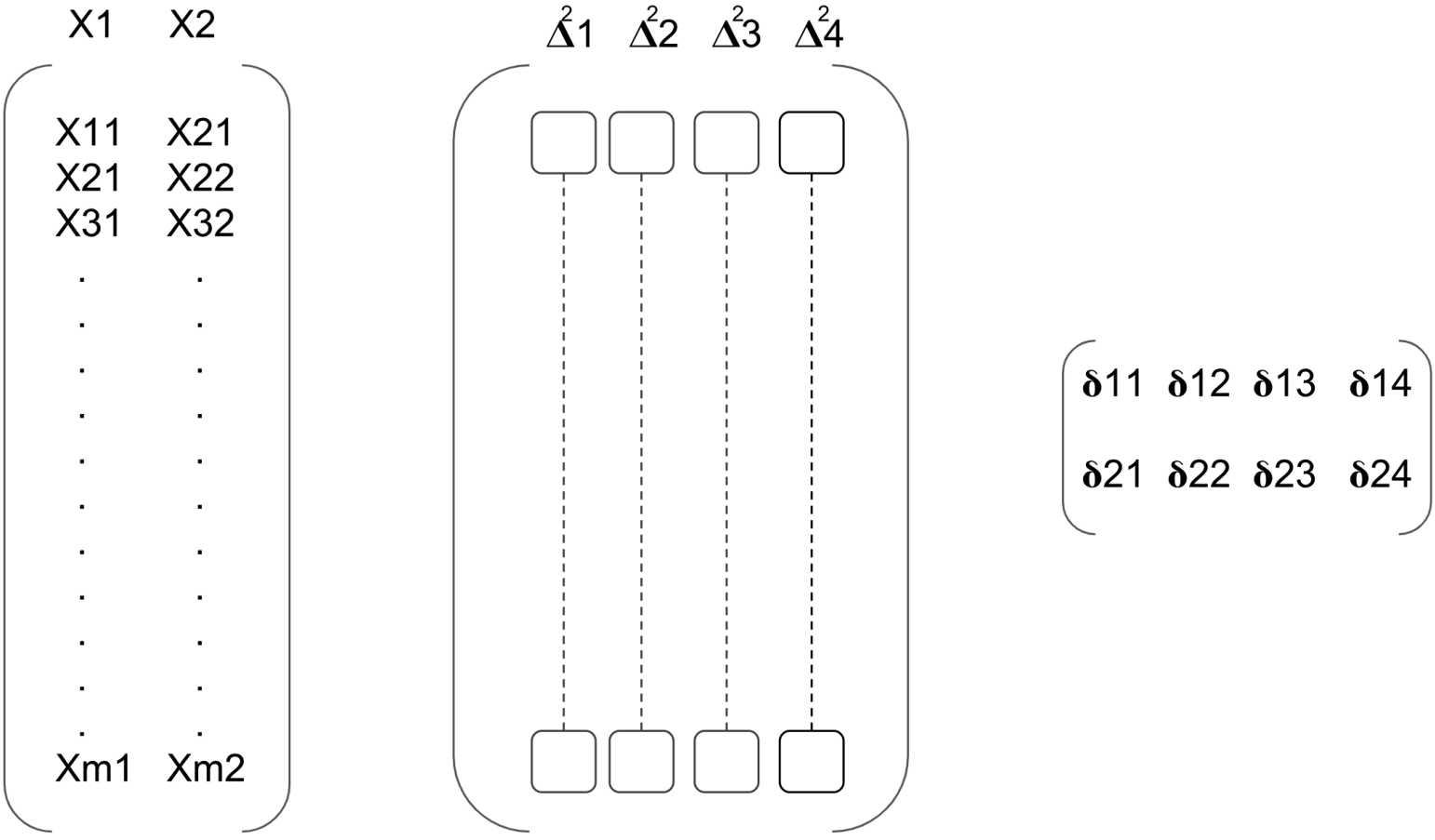




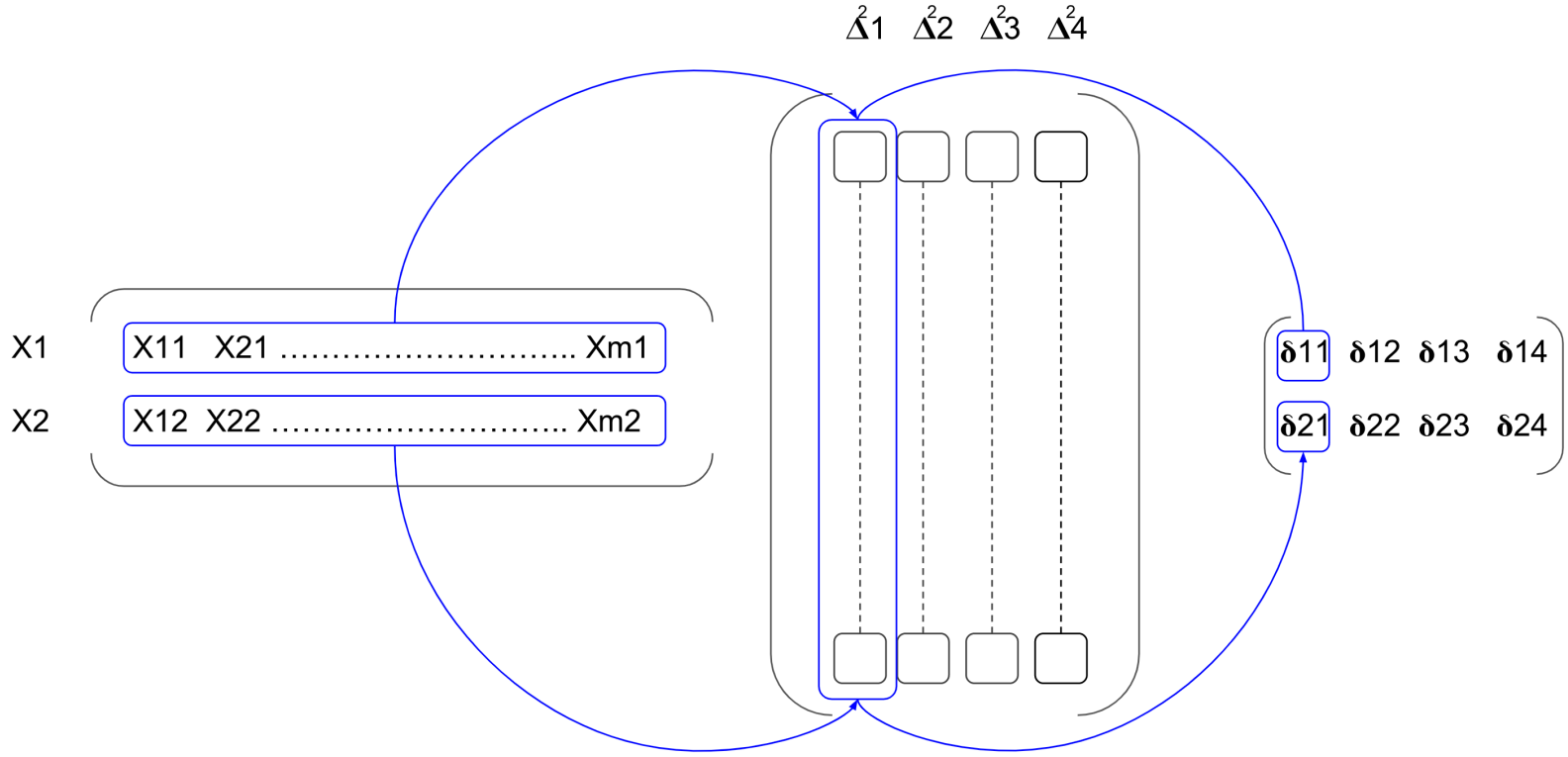
Updating Weights

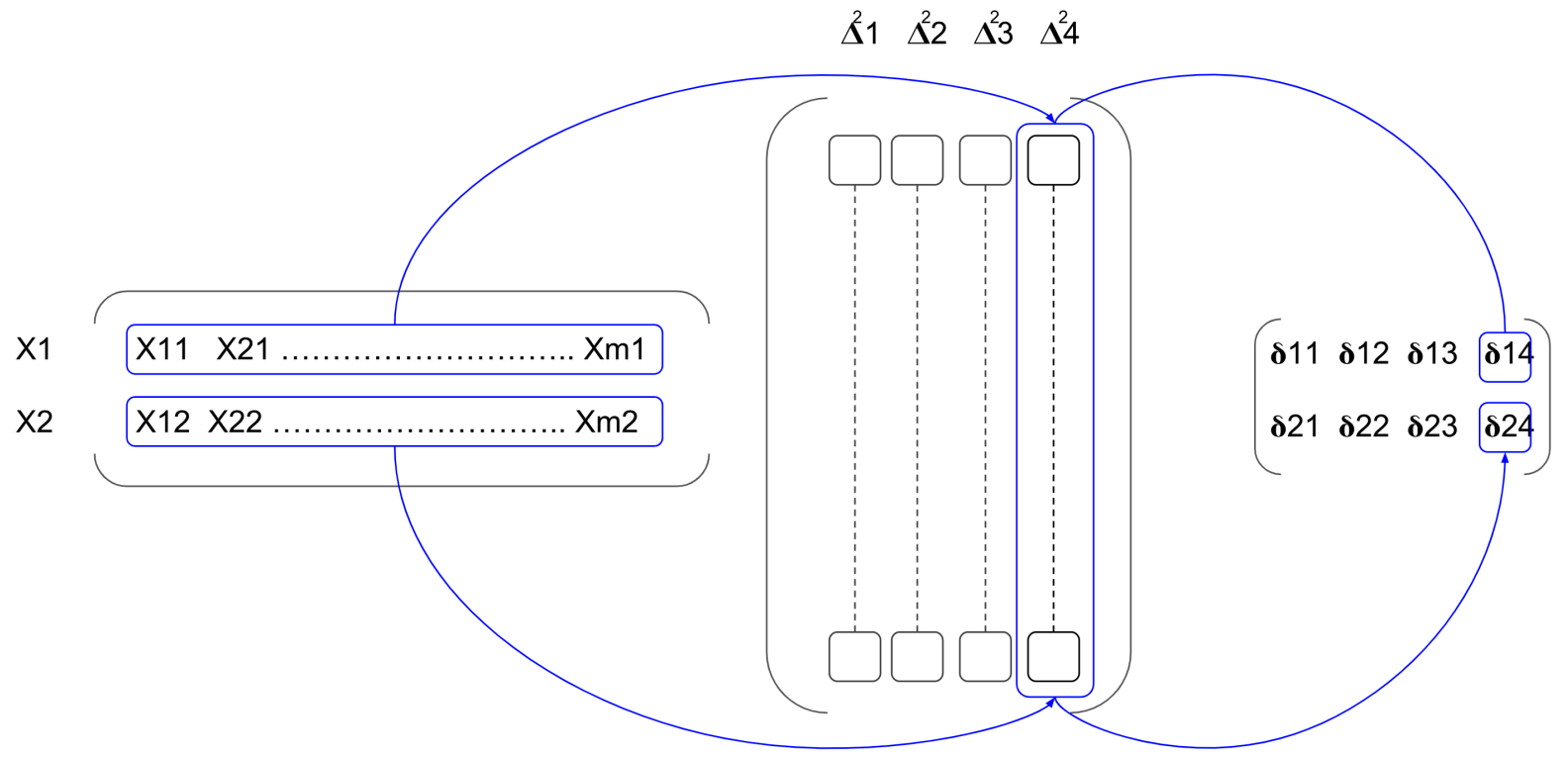
Updating the weights at layer ***i***, involves getting the input at the layer, perform the dot product with the delta of the layer then use the result to update the weights.

If we take layers 1 and 2 of our example, we have an input, at layer 1, with dimension (m, 2) and the dimension of delta at layer 2 (m, 4). However the weights matrix connecting layer 1 and 2 has the dimension (2, 4).



To be able to update the weights matrix, the dot product between X and **Δ** should result in a (2, 3) matrix. This is done by getting the transpose of X that is X^T, then perform the dot product X^T **.** **Δ** that we call ***δ***.





Now we can update the weight matrix using the gradient descent formula
$W^{n+1} = W^n - \alpha * \delta$ (in here n is the version of W and not the layer index)

Conclusion

Chances that you do the back propagation from scratch are slim, thanks to the multitude of libraries that already do that. However if you want to do the exercice yourself, you should pay attention to the dimensions, otherwise you will fall into a labyrinth of computational errors.

Source Code

The source code of the XOR example developed in the previous article is included below, it uses matrix of sample instead of one sample at a time.

```
import numpy as np


def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))


def sigmoid_prime(x):
    return sigmoid(x) * (1.0 - sigmoid(x))


def tanh(x):
    return np.tanh(x)


def tanh_prime(x):
    return 1.0 - np.tanh(x) ** 2

def relu(x):
    return np.maximum(x,0)


def relu_prime(x):
    x[x <= 0] = 0
    x[x > 0] = 1
    return x

class Layer:

    def __init__(self, dim, id, act, act_prime, isoutputLayer = False):
        self.weight = 2 * np.random.random(dim) - 1
        self.delta = None
        self.A = None
        self.activation = act
        self.activation_prime = act_prime
        self.isoutputLayer = isoutputLayer
        self.id = id


    def forward(self, x):
        z = np.dot(x, self.weight)
        self.A = self.activation(z)
        self.dZ = np.atleast_2d(self.activation_prime(z));

        return self.A

    def backward(self, y, rightLayer):
        if self.isoutputLayer:
```

```
            error = self.A - y
            self.delta = np.atleast_2d(error * self.dZ)
        else:
            self.delta = np.atleast_2d(
                np.dot(rightLayer.delta, rightLayer.weight.T)
                * self.dZ)
        return self.delta

    def update(self, learning_rate, left_a):
        a = np.atleast_2d(left_a)
        d = np.atleast_2d(self.delta)
        ad = a.T.dot(d)
        self.weight -= learning_rate * ad


class NeuralNetwork:

    def __init__(self, layersDim, activation='tanh'):
        if activation == 'sigmoid':
            self.activation = sigmoid
            self.activation_prime = sigmoid_prime
        elif activation == 'tanh':
            self.activation = tanh
            self.activation_prime = tanh_prime
        elif activation == 'relu':
            self.activation = relu
            self.activation_prime = relu_prime

        self.layers = []
        for i in range(1, len(layersDim) - 1):
            dim = (layersDim[i - 1] + 1, layersDim[i] + 1)
            self.layers.append(Layer(dim, i, self.activation, self.activation_prime))

        dim = (layersDim[i] + 1, layersDim[i + 1])
        self.layers.append(Layer(dim, len(layersDim) - 1, self.activation, self.activation_prime, True))

    def fit(self, X, y, learning_rate=0.1, epochs=10000):
        # Add column of ones to X
        # This is to add the bias unit to the input layer
        ones = np.atleast_2d(np.ones(X.shape[0]))
        X = np.concatenate((ones.T, X), axis=1)



        for k in range(epochs):



            a=X
```

```
            for l in range(len(self.layers)):
                a = self.layers[l].forward(a)


            delta = self.layers[-1].backward(y, None)

            for l in range(len(self.layers) - 2, -1, -1):
                delta = self.layers[l].backward(delta, self.layers[l+1])



            a = X
            for layer in self.layers:
                layer.update(learning_rate, a)
                a = layer.A

    def predict(self, x):
        a = np.concatenate((np.ones(1).T, np.array(x)), axis=0)
        for l in range(0, len(self.layers)):
            a = self.layers[l].forward(a)
        return a



if __name__ == '__main__':

    X = np.array([[0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1],
                  [0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1],
                  [0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1]
    ])

    y = np.array([[0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0]]).T


    # tanh
    nn = NeuralNetwork([2, 3, 5, 8, 1], activation='tanh')

    nn.fit(X, y, learning_rate=0.1, epochs=10000)
```

```
print("\n\nResult with tanh")
for e in X:
    print(e, nn.predict(e))



#  sigmoid
nn = NeuralNetwork([2, 3, 4, 1], activation='sigmoid')

nn.fit(X, y, learning_rate=0.3, epochs=20000)

print("\n\nResult with sigmoid")
for e in X:
    print(e, nn.predict(e))



#  relu
nn = NeuralNetwork([2, 3, 4, 1], activation='relu')

nn.fit(X, y, learning_rate=0.1, epochs=50000)

print("\n\nResult with relu")
for e in X:
    print(e, nn.predict(e))
```