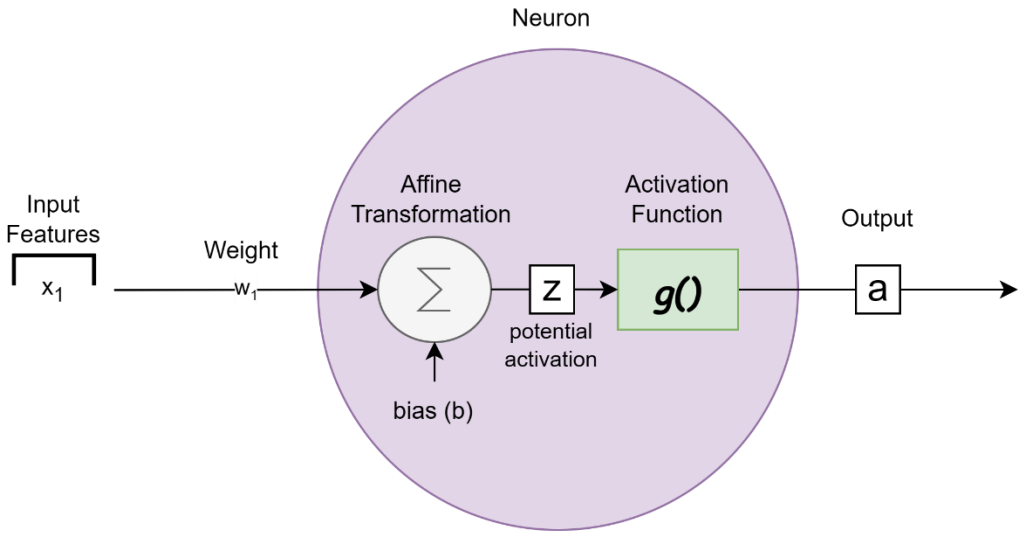
Artificial Neural Network

1 MATHEMATICAL MODEL OF NEURON



Consider a neuron with only one input features. Let w_1 be the weight of the input. Here, x_1 refers the input and b is the bias to the network.

$$z = w_1.x_1 + b$$

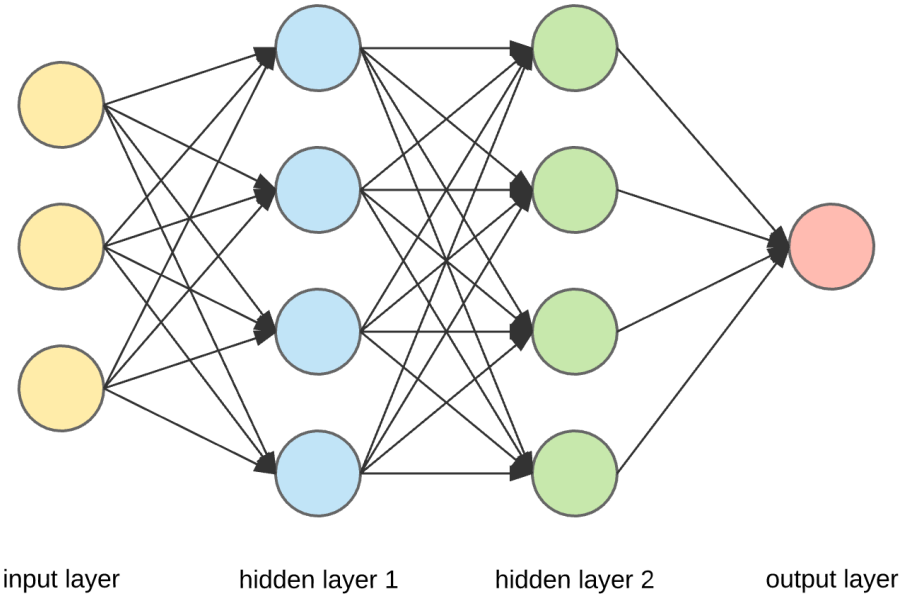
Where z is the activation potential. Now, the activation potential goes through the activation function

$$a = g(z)$$

Where, $g()$ is the activation function and a is the activation value which is also the output of the neuron.

2 FEED FORWARD NEURAL NETWORK

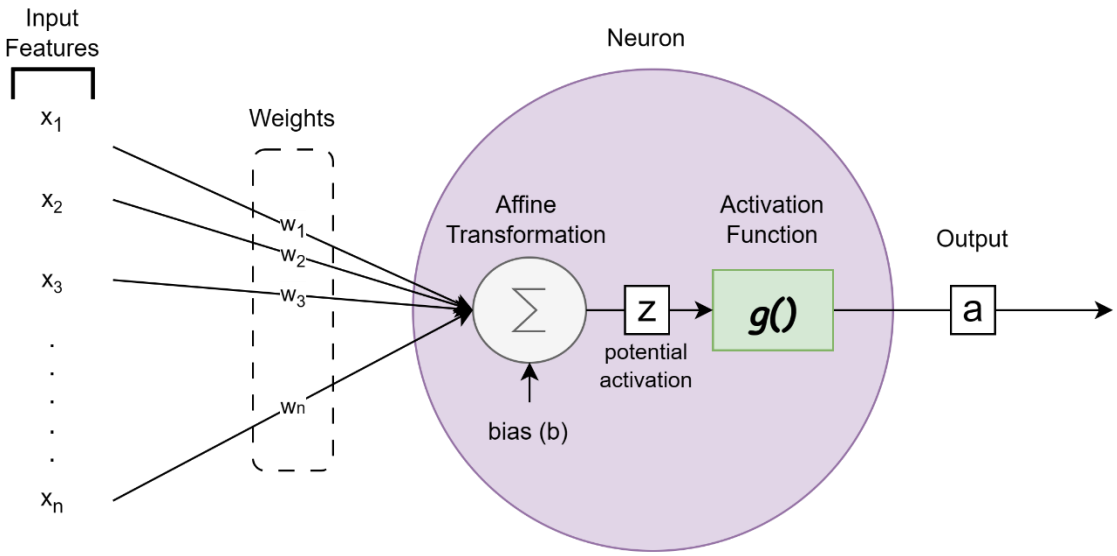
Neural Network is the network of neurons arranged in a structure such that there are multiple neurons in each layer of network. There is a connection from neuron of one layer of network to another layer of network.



2.1 FORWARD PROPAGATION

Forward propagation is the computational execution of neural network from input to the output. We now construct a neural network step by step increasing the complexity in the network. Let's get started with single neuron but with multiple inputs.

Step 1: Single layered, multiple inputs



Consider a single neuron that has multiple input features $x_1, x_2, x_3, \ldots x_n$ then,

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

*Notation: x is a scaler and **bolded $\boldsymbol{x}$ is a vector.***

And the weight associated with each input features is $w_1, w_2, w_3, \ldots, w_n$ then, we have weight vectors as:

$$w = \begin{bmatrix} w_1 & w_2 & w_3 & w_4 \end{bmatrix}$$

Later we have to take dot product of $\boldsymbol{w}$ and $\boldsymbol{x}$, so for our ease we take w as row vector. If you are comfortable with $\boldsymbol{w^T}.\boldsymbol{x}$, feel free to use was column vector so.

Just remember, throughout this document, we will be using $\boldsymbol{w}.\boldsymbol{x}$ for the dot product and $\boldsymbol{W}.\boldsymbol{X}$ for matrix multiplication avoiding the use of transpose.

Now, for the computation that occurs on neuron, we first compute activation potential through affine transformation as:

$$z = \boldsymbol{w}.\boldsymbol{x} + b$$

Here, **w** and $\boldsymbol{x}$ are the vectors. But b is a bias occurring in a neuron and has no relation with number of input so its always scalar. And, the final output z would be a scalar (How? Check yourself).

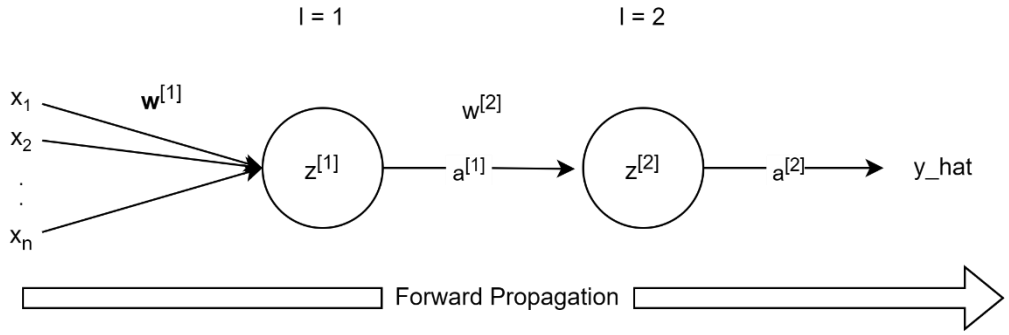
Now, applying activation function on activation potential, we get activation value:

$$a = g(z)$$

Here, a is the activation value and is scalar again, inherited form z.

Step 2: Single neuron in each layer for two-Layered network

Now, let's construct a neural network such that there are multiple layers where each layer contains only one neuron:



Then the forward computation in the first layer $l = 1$ would be:

$$z^{[1]} = \boldsymbol{w}^{[\mathbf{1}]}.\boldsymbol{x} + b^{[1]}$$

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

**Remember, Superscript [1] refers to the layer we are computing in but not the power raised.*

Here, $\boldsymbol{x}$ and $\boldsymbol{w}$ are vectors whereas b is scalar. Also, z and a are scalars. Each layer in the neural network is likely to use different activation function thus we are using $g^{[1]}$. Thus,

$\boldsymbol{x}$ is the input vector

$\boldsymbol{w}^{[\mathbf{1}]}$ is the weight vector of layer 1

$b^{[1]}$ is the bias in the neuron of layer 1

$z^{[1]}$ is the activation potential in the neuron of layer 1

$a^{[1]}$ is the activation value in the neuron of layer 1

And, $g^{[1]}$ is the activation function in layer 1

Now for computation in 2nd layer, we now use the activation value from layer 1 as input and associate new weight to this input. Thus, in the layer l = 2 computation would be:

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

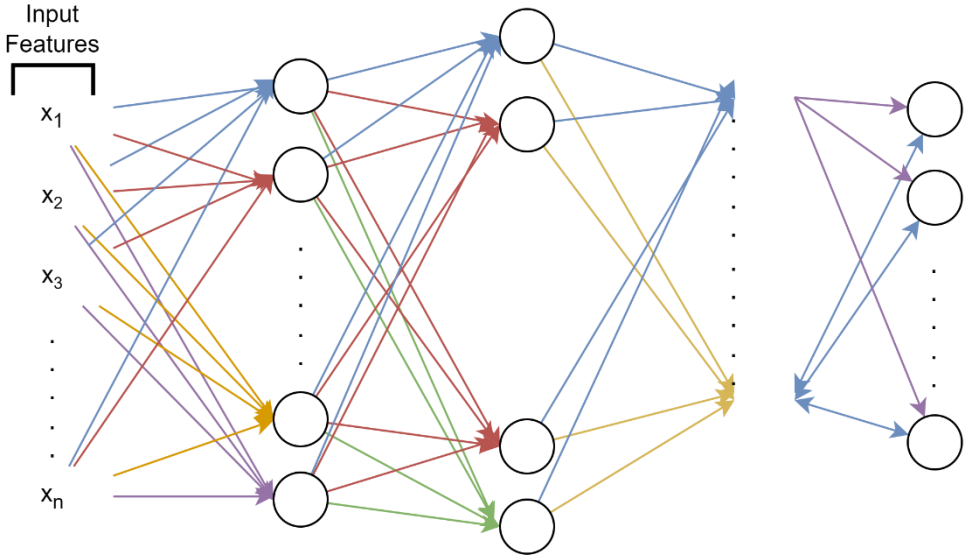
$$a^{[2]} = g^{[2]}(z^{[2]})$$

The activation value of second layer is the output of the neural network (*hence, we can only use Sigmoid or SoftMax activation function in the output layer if the task is classification otherwise, we can use any activation function including linear function if the task is regression*). Then,

$$\hat{y} = a^{[2]}$$

Step 3: Multiple neurons in each layer for two-Layered network

Now, let's construct a neural network such that there are multiple layers and each layer contains multiple neurons:



L-layered Network

Then the forward computation in the first layer $l = 1$ would be:

For each neuron $j = 1 \ldots n^{[l]}$,

$$\boldsymbol{z}_j^{[1]} = \boldsymbol{w}_j^{[1]}.\boldsymbol{x} + b_j^{[1]}$$

$$\boldsymbol{a}_j^{[1]} = g^{[1]}(\boldsymbol{z}_j^{[1]})$$

Similarly, for the layer $l = 2$, for each neuron $j = 1 \ldots n^{[l]}$,

$$\boldsymbol{z}_j^{[2]} = \boldsymbol{w}_j^{[2]}.\boldsymbol{a}^{[1]} + b_j^{[2]}$$

$$\boldsymbol{a}_j^{[2]} = g^{[2]}(\boldsymbol{z}_j^{[2]})$$

And so on for all the layers.

Forward Propagation

Thus, the forward propagation is of the form:

$For\ l\ = 1 \ldots L\ \{$

$\quad For\ each\ neuron\ j\ =\ 1 \ldots n^{[l]}\ \{$

$\quad\quad z_j^{[l]} = \boldsymbol{w}_j^{[l]}.\boldsymbol{a}^{[l-1]} + b_j^{[l]}$

$\quad\quad a_j^{[l]} = g^{[l]}(z_j^{[l]})$

$\quad \}$

$\}$

Step 4: Vectorization

We form a matrix and vector from weights and biases of neuron of a layer. And, instead of operating individually on each component, we operate directly in the form of vector and matrix. This way, we are able to perform all the computation at once.

For example,

For layer $l\ =\ 1$, the weight input in each neuron is already a vector. So, for $n^{[1]}$ neuron in the first layer of network, we will have the same number of Weight vectors incoming from all the inputs features i.e. each weight vector has a size of $1 \times n$, where n is the size of input features. And if we bring together all $n^{[1]}$ number of vectors into a matrix, we get a size of $n^{[1]} \times n$.

Similarly, each neuron has a single bias but if we stack the biases of all neurons in the first layer, we get a bias vector $n^{[1]} \times 1$.

$$\boldsymbol{z}^{[1]} = \boldsymbol{W}^{[1]}.\boldsymbol{x} + \boldsymbol{b}^{[1]}$$

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

Here,

input $\boldsymbol{x}$ is a vector of size $n \times 1$,

$\boldsymbol{W}$ is a weight matrix of size $n^{[1]} \times n$ and

$\boldsymbol{b}$ is a bias vector of size $n^{[1]} \times 1$

Thus $\boldsymbol{W}.\boldsymbol{x}$ produces a vector of size $n^{[1]} \times 1$ and addition with bias b produces the activation potential vector $\boldsymbol{z}$ of size $n^{[1]} \times 1$. We then obtain activation vector $\boldsymbol{a}$ of size $n^{[1]} \times 1$.

The reason $\boldsymbol{b}$, $\boldsymbol{z}$ and $\boldsymbol{a}$ are vectors and **W** a matrix now is we are taking all the neurons of layer at once in the operation to avoid iterative computation. This let us to compute parallelly for all the neurons in layer.

Vectorized Forward Propagation

Now, the vectorized form of forward propagation is:

$$For\ l\ = 1 \ldots L\ \{$$
$$\boldsymbol{z}^{[l]} = \boldsymbol{W}^{[l]}.\boldsymbol{a}^{[\boldsymbol{l-1}]} + \boldsymbol{b}^{[l]}$$
$$\boldsymbol{a}^{[l]} = g^{[l]}(\boldsymbol{z}^{[l]})$$
$$\}$$

Step 5: Multiple training examples,

We were taking single training examples with n features as of now. Let's consider we have m training examples in our dataset and we take all training examples at once. Then our input would be a matrix of size $n \times m$ and represented by $\boldsymbol{X}$. And as earlier, for layer 1

$\boldsymbol{W}$ is a weight matrix of size $n^{[1]} \times n$ and

$\boldsymbol{b}$ is a bias vector of size $n^{[1]} \times 1$

Then, $\boldsymbol{W}.\boldsymbol{X}$ produces a vector of size $n^{[1]} \times m$ and addition with bias $\boldsymbol{b}$ produces the activation potential vector $\boldsymbol{Z}$ of size $n^{[1]} \times m$. We then obtain activation vector $\boldsymbol{A}$ of size $n^{[1]} \times m$. Now, we can proceed in the same till the last layer.

Vectorized Forward Propagation

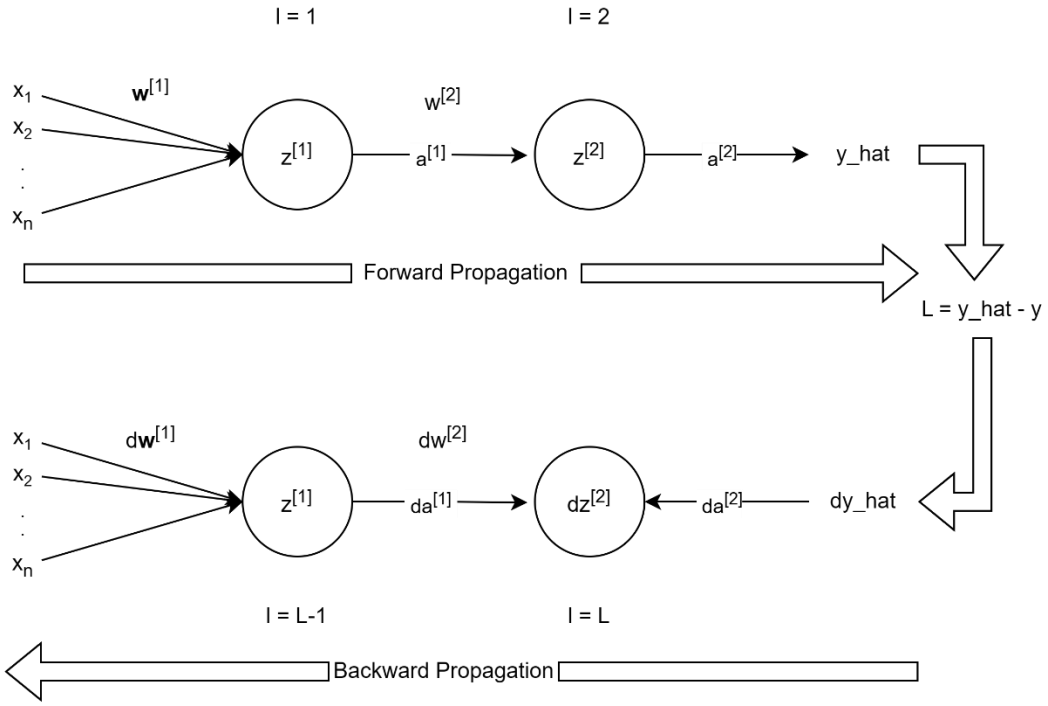
Now, the vectorized form of forward propagation becomes:

$$For\ l\ = 1 \dots L\ \{$$
$$\boldsymbol{Z^{[l]} = W^{[l]}.A^{[l-1]} + b^{[l]}}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$
$$\}$$

This concludes the forward propagation step by step derivation.

2.2 BACKWARD PROPAGATION

To understand and derive the basic form of back propagation, consider a neural network as below where each layer in the neural network contains only one neuron and we are taking only two layers in the network:



To make our derivation easy, we consider 2nd layer as L layer and 1st layer as L-1 layer.

Before proceeding to the backward propagation, let's devise the form of forward propagation in each layer:

In layer L-1,

$$z^{[L-1]} = w^{[L-1]}a^{[L-2]} + b^{[L-1]}$$

$$a^{[L-1]} = g^{[L-1]}(z^{[L-1]})$$

**Note: We use small case notation for z, w and a which means we are computing for the single neuron in the particular layer.*

Similarly, in layer L,

$$z^{[L]} = w^{[L]}a^{[L-1]} + b^{[L]}$$

$$a^{[L]} = g^{[L]}(z^{[L]})$$

Since L-th layer is the final layer in our network,

$$\hat{y} = a^{[L]}$$

And, at the end of forward computation we compute the final output of the neural network as above. Then we compute error for each training example, and, taking all training examples we generally compute an average of errors known as **cost function**.

For regression, cost function can be

$$C = \frac{\sum_{i=1}^{m}(\hat{y}_i - y_i)}{m}$$

And for classification, cost function can be

$$C = \frac{1}{m}\sum_{i=1}^{m}[-y_i\, log\hat{y}_i - (1 - y_i)\log(1 - \hat{y}_i)]$$

Where, C is the cost function.

Depending on the nature of problem, we may take any form of cost function.

We can now start the backpropagation on neural network. Through backpropagation we propagate the error devised from the forward propagation in the network by comparing actual value (y) with the predicated value ($\hat{y}$) .

We propagate the error backward in the network so as to update the weight and biases which were input in the neuron. And using gradient descent, we can obtain the new weight and bias for each neuron.

So, let's get started with the backpropagation step by step:

Step 1: First compute the derivate of Error with respect to the $\hat{y}$ because error is computed using y and $\hat{y}$ and y is always the same thus constant for lifetime.

$$\frac{\partial C}{\partial \hat{y}}$$

Derivation of this is totally based on the form of the cost function we use to evaluate the problem. Thus, we won't compute any further derivative of it.

But, to ease on writing notation we will write,

$$d\hat{y} = \frac{\partial C}{\partial \hat{y}} \dots\dots\dots (1)$$

Similarly, for any future computation, if derivative of Cost is computed w.r.t. say p variable, we denote by dp.

$$dp = \frac{\partial C}{\partial p}$$

Now, we compute the derivative in Layer L as below:

Step 2: We compute the derivative of cost with respect to the $a^{[L]}$,

$$da^{[L]} = \frac{\partial C}{\partial a^{[L]}} = \frac{\partial C}{\partial \hat{y}} = d\hat{y} \dots\dots\dots (2)$$

Step 3: Compute the derivative of cost with respect to the $z^{[L]}$,

$$dz^{[L]} = \frac{\partial C}{\partial z^{[L]}} = \frac{\partial C}{\partial a^{[L]}} . \frac{\partial a^{[L]}}{\partial z^{[L]}} = da^{[L]} . a'^{[L]} \dots\dots\dots (3)$$

Here,

$\frac{\partial a^{[L]}}{\partial z^{[L]}} = a'^{[L]}$ because, a is activation value obtained through applying activation function on z. And, depending on activation function, the derivative value will be different. Thus, we denote this by $a'^{[L]}$ in general.

For example, for sigmoid function

$$g'(z) = g(z)(1 - g(z))$$

And, for RELU

$$g'(z) = \begin{cases} 1 \ if \ z \geq 0 \\ 0 \ if \ z < 0 \end{cases}$$

Step 4: Compute the derivative of cost w.r.t. $w^{[L]}$.

$$dw^{[L]} = \frac{\partial C}{\partial w^{[L]}} = \frac{\partial C}{\partial z^{[L]}} . \frac{\partial z^{[L]}}{\partial w^{[L]}} = dz^{[L]} . a^{[L-1]} \quad [\because We\ obtained\ this\ form\ (3)] \ \ldots\ldots\ldots (4)$$

Step 5: Compute the derivative of cost w.r.t. $b^{[L]}$

$$db^{[L]} = \frac{\partial C}{\partial b^{[L]}} = \frac{\partial C}{\partial z^{[L]}} . \frac{\partial z^{[L]}}{\partial b^{[L]}} = dz^{[L]} . 1 = dz^{[L]} \quad [\because z = wx + b\ and\ from\ (3)] \ldots\ldots\ldots (5)$$

Now, we compute the derivative in Layer L-1 as below:

Step 6: Compute the derivative of cost w.r.t. $a^{[L-1]}$.

$$da^{[L-1]} = \frac{\partial C}{\partial a^{[L-1]}} = \frac{\partial C}{\partial z^{[L]}} . \frac{\partial z^{[L]}}{\partial a^{[L-1]}} = dz^{[L]} . w^{[L]} \quad [\because from\ 3] \ldots\ldots\ldots (6)$$

Remember,

The sequence of derivatives we performed is based on the forward propagation form in L-th layer:

$$z^{[L]} = w^{[L]} a^{[L-1]} + b^{[L]}$$

$$a^{[L]} = g^{[L]}(z^{[L]})$$

This is because backpropagation uses a chain method for computing derivative.

Step 7: Next, compute the derivative of cost w.r.t. $z^{[L-1]}$

$$dz^{[L-1]} = \frac{\partial C}{\partial z^{[L-1]}} = \frac{\partial C}{\partial a^{[L-1]}} . \frac{\partial a^{[L-1]}}{\partial z^{[L-1]}} = da^{[L-1]} . a'^{[L-1]} \ [\because from\ (6)] \ldots\ldots\ldots (7)$$

Step 8: Again, we compute the derivative of cost w.r.t. $w^{[L-1]}$ and $b^{[L-1]}$ similar as above:

$$dw^{[L-1]} = \frac{\partial C}{\partial w^{[L-1]}} = \frac{\partial C}{\partial z^{[L-1]}} . \frac{\partial z^{[L-1]}}{\partial w^{[L-1]}} = dz^{[L-1]} . a^{[L-2]} \ \ldots\ldots\ldots (8)$$

And,

$$db^{[L-1]} = \frac{\partial C}{\partial b^{[L-1]}} = \frac{\partial C}{\partial z^{[L-1]}} . \frac{\partial z^{[L-1]}}{\partial b^{[L-1]}} = dz^{[L-1]} . 1 = dz^{[L-1]}$$
$$= a^{[L]} . a'^{[L]} . w^{[L]} . a'^{[L-1]} \quad \ldots\ldots\ldots (9)$$

And so on for other layers like L-2, L-3… to 1 if they exist.

Now, if we compare the form of backpropagation derivative in weight (w) and bias (b) of L layer and L-1 layer, we notice a pattern.

From above derivations, we can write:

For Layer L

$$dz^{[L]} = da^{[L]} . a'^{[L]} = d\hat{y} . a'^{[L]}$$

$$dw^{[L]} = dz^{[L]} . a^{[L-1]}$$

$$db^{[L]} = dz^{[L]}$$

For layer L-1

$$da^{[L-1]} = dz^{[L]} . w^{[L]}$$

$$dz^{[L-1]} = da^{[L-1]} . a'^{[L-1]} = dz^{[L]} . w^{[L]} . a'^{[L-1]}$$

$$dw^{[L-1]} = dz^{[L-1]} . a^{[L-2]}$$

$$db^{[L-1]} = dz^{[L-1]}$$

And, if layer L-2 exists, we can write,

$$dz^{[L-2]} = dz^{[L-1]} . W^{[L-1]} . a'^{[L-2]}$$

$$dw^{[L-2]} = dz^{[L-2]}.a^{[L-3]}$$

$$db^{[L-2]} = dz^{[L-2]}$$

Back Propagation

So, in easy term, we can write

For $l = L$ i.e. last layer

$$dz^{[L]} = d\hat{y}.a'^{[L]}$$

$$dw^{[L]} = dz^{[L]}.a^{[L-1]}$$

$$db^{[L]} = dz^{[L]}$$

for $l = L - 1\ to\ 1$ i.e. except last layer

$$dz^{[l]} = dz^{[l+1]}.w^{[l+1]}.a'^{[l]}$$

$$dw^{[l]} = dz^{[l]}.a^{[l-1]}$$

$$db^{[l]} = dz^{[l]}$$

But, in general neural network, we won't have single neuron in each layer. Instead, only the L layer depicts the form as above, and from L-1 to 1st layer depicts form similar to below:

Following two case arises from the above:

1. From Lth layer, the error is back propagated to all the neuron L-1 layer and from L-1 layer to L-2 layer and so on. The back propagated error from l^{th} is passed as a whole to each neuron of $(l-1)$ layer.
2. Also, each neuron in L-1 layer receives back propagated error from each neuron in L layer as shown in above diagram i.e. each neuron in L-1 layer accumulates error from all the neuron in Lth layer. So, we can say,

   For neuron i, in layer $l = L - 1 \ldots 1$,

$$\sum_{j=1}^{n^{[l]}} dz_j^{[l+1]}.w_{ij}^{[l+1]}$$

is the accumulated error through the back propagation. This changes the form of equations we derived above except the last layer as it remains same. Here $n^{[l]}$ is the number of neurons in that particular layer.

Backpropagation Formula (*Memorize*)

Finally, we get our required form for back propagation in iterative form as:

For $l = L$ i.e. last layer, for each neuron j

$$dz^{[L]} = d\hat{y}.a'^{[L]}$$

$$dw^{[L]} = dz^{[L]}.a^{[L-1]}$$

$$db^{[L]} = dz^{[L]}$$

for $l = L - 1\ to\ 1$ i.e. except last layer, for each neuron j

$$dz^{[l]} = a'^{[l]} \sum_{j=1}^{n^{[l]}} dz_{ij}^{[l+1]}.w_{ij}^{[l+1]}$$

$$dw^{[l]} = dz^{[l]}.a^{[l-1]}$$

$$db^{[l]} = dz^{[l]}$$

Now, we can apply weight update rule using gradient descent.

$$w = w - \alpha\frac{\partial C}{\partial w} = w - \alpha.dw$$

$$b = b - \alpha\frac{\partial C}{\partial b} = b = \alpha.db$$

Back Propagation Algorithm:

1. Randomly initialize the weight and bias of all neurons in the network.
2. Repeat until convergence:

   for each training tuple **x** in X,

   a. forward propagation

      For each layer l in the network for each neuron j, compute

$$z = \boldsymbol{w}^{[l]}\boldsymbol{x} + b$$

$$a = g^{[l]}(z)$$

b. backward propagation

For $l = L$ i.e. last layer, for each neuron j

$$dz^{[L]} = d\hat{y}.a'^{[L]}$$

$$dw^{[L]} = dz^{[L]}.a^{[L-1]}$$

$$db^{[L]} = dz^{[L]}$$

for $l = L - 1\ to\ 1$ i.e. except last layer, for each neuron j

$$dz^{[l]} = a'^{[l]} \sum_{j=1}^{n^{[l]}} dz_{ij}^{[l+1]}.w_{ij}^{[l+1]}$$

$$dw^{[l]} = dz^{[l]}.a^{[l-1]}$$

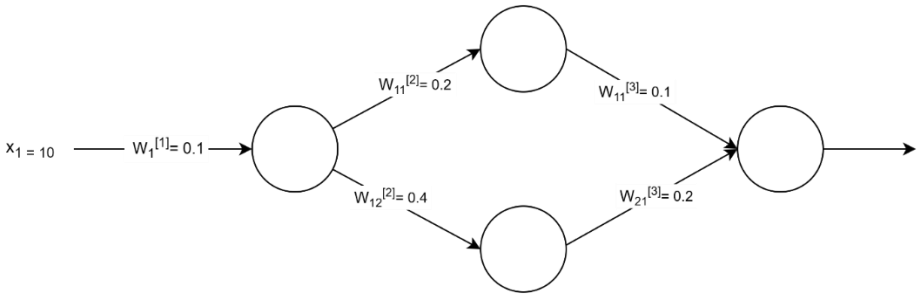
$$db^{[l]} = dz^{[l]}$$

3. Update weight and biases of each neuron for all layer in the network:

$$w^{[l]} = w^{[l]} - \alpha.dw^{[l]}$$

$$b^{[l]} = b^{[l]} - \alpha.db^{[l]}$$

i) **Numerical Problem 1:** Given a neural network, perform forward propagation considering a classification problem and compute the loss. Consider a sigmoid activation function for all the layers.



Where (x, y) = (10, 0).

Solution:

There are three layers in the network. And, at first let's execute forward propagation. Since there is only one training example, we use a vector form and for our ease on calculation we ignore bias. Thus,

In layer 1:

w is a scaler as we have only one input thus, $w = 0.1$. And $x = a^{[0]}$.

$$a^{[1]} = sigmoid(wa^{[0]}) = 1 * 0.2 = sigmoid(0.2) = 0.55$$

In layer 2:

$\boldsymbol{w}$ is a vector as we have two neurons in layer 2. Thus,

$$\boldsymbol{w} = \begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix}$$

$$\boldsymbol{a^{[2]}} = sigmoid(\boldsymbol{w}a^{[1]}) = sigmoid(\boldsymbol{w}.a^{[\mathbf{1}]}) = sigmoid\left(\begin{bmatrix} 0.2 \\ 0.4 \end{bmatrix}.(0.55)\right)$$

$$= sigmoid\left(\begin{bmatrix} 0.2 * 0.55 \\ 0.4 * 0.55 \end{bmatrix}\right) = sigmoid\left(\begin{bmatrix} 0.11 \\ 0.22 \end{bmatrix}\right) = \begin{bmatrix} 0.53 \\ 0.55 \end{bmatrix}$$

Now, in layer 3:

w is again a vector as two inputs are there for a single neuron. Thus

$$w = [0.1 \quad 0.2]$$

$$a^{[3]} = sigmoid(\boldsymbol{w}.\boldsymbol{a^{[2]}}) = sigmoid([0.1 \quad 0.2].\begin{bmatrix} 0.53 \\ 0.55 \end{bmatrix})$$

$$= sigmoid(0.1 * 0.53 + 0.2 * 0.55) = sigmoid(0.163) = 0.54$$

The final output of the network is 0.54.

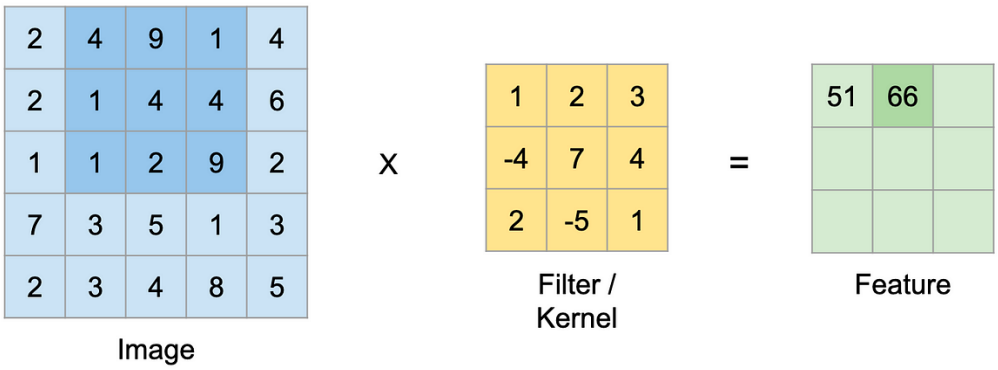
Now, computing the loss using log loss function

$$C = -ylog\hat{y} - (1-y)\log(1-\hat{y})$$

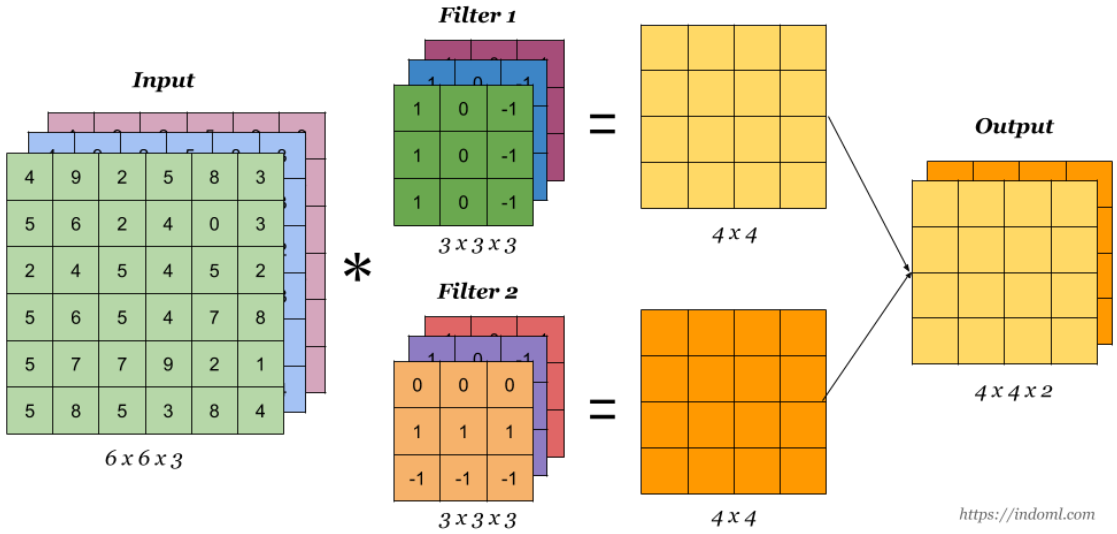
$$= -(0 * \log(0.54)) - (1-0) * log * (1-0.54) = 0.337242168$$

3 CONVOLUTIONAL NEURAL NETWORK

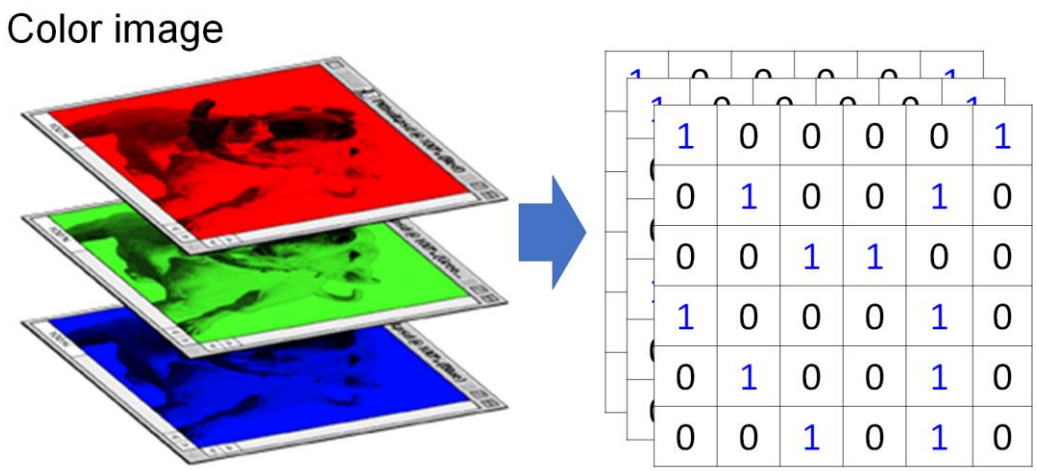
Convolutional Neural Network is a type of neural network with convolutional layers in the network. A convolutional layer has a number of filters (*a.k.a kernels*) that are used in convolution operation. A convolution operation focuses on extracting/preserving important features from the input.



Filters are used to detect a pattern from certain region, in above case at a time small pattern from 3×3 region of input image is detected. There can be multiple filters applied at a time:

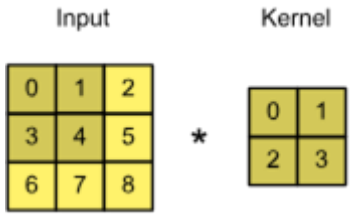


If the image is grayscale, then the image is considered as a matrix, each value in the matrix ranges from 0-255. We can even normalize these values let's say in range 0-1. 0 represents white and 1 represents black. On the other hand, If the image is colored, then three matrices representing the RGB colors with each value in range 0-255.



3.1 CONVOLUTION OPERATION

Convolution Operation is actually misnomer, since the operations they express are more accurately described as cross-correlations. Consider a two-dimensional cross correlation operation as below:



We begin with kernel window positioned to the upper left corner and perform convolution operation as

$$0 * \ 0 + 1 * 1 + 3 * 2 + 4 * 3 = 0 + 1 + 6 + 12 = 19$$

This is the elementwise multiplication of two matrices and summation of the values.

Next the filter window slides by 1 and the convolution operation is conducted as:

$$1 * 0 + 2 * 1 + 4 * 2 + 5 * 3 = 0 + 2 + 8 + 15 = 25$$

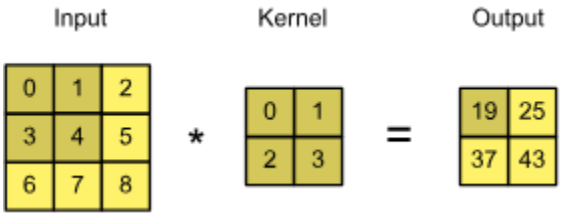
After, sliding the filter fully on horizontal direction we slide the filter by 1 vertically and start again from the left-hand side

$$3 * 0 + 4 * 1 + 6 * 2 + 7 * 3 = 0 + 4 + 12 + 21 = 37$$

And slide horizontally by 1, we get

$$4 * 0 + 5 * 1 + 7 * 2 + 8 * 3 = 0 + 5 + 14 + 24 = \ 43$$

Thus, the final output would be:



This means the output matrix has reduced size. This can be explained by:

$$(n_h - f_h + 1) \times (n_w - f_w + 1)$$

Where,

$n_h = height\ of\ an\ image\ i.e.no.of\ pixel\ vertically\ arranged\ in\ image.$

$n_w = width\ of\ an\ image\ i.e.no.of\ pixel\ horizontally\ arranged\ in\ image.$

$f_h = height\ of\ filter\ i.e.no.of\ pixel\ vertically\ arranged\ in\ filter$

$f_w = width\ of\ filter\ i.e.no.of\ pixel\ horizontally\ arranged\ in\ image$

Numerical: Consider a grey scale image of 6×6 size,

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

Apply following two filters:

1	-1	-1
-1	1	-1
-1	-1	1

-1	1	-1
-1	1	-1
-1	1	-1

Solution:

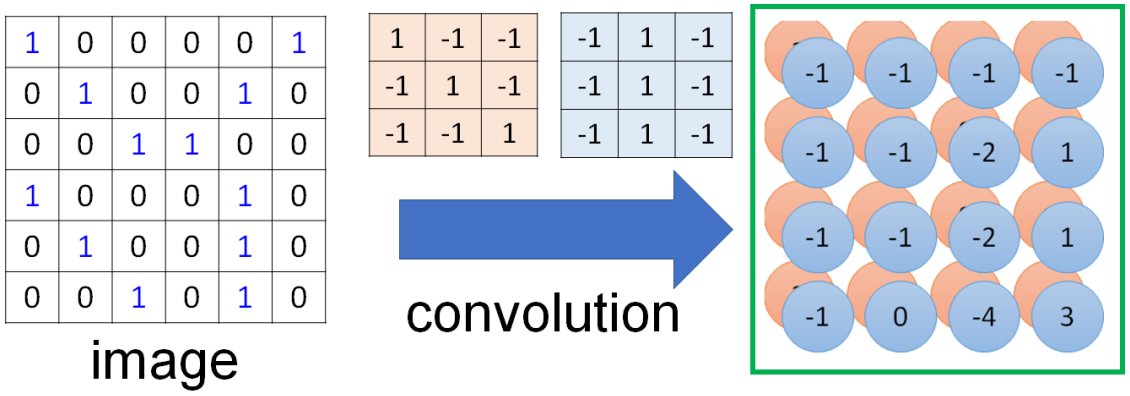
Applying first filter,

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

Applying second filter,

-1	-1	-1	-1
-1	-1	-2	1
-1	-1	-2	1
-1	0	-4	3

Thus, we get two different images by applying two filters. This means, applying multiple filters in a stack produces multiple output images.



Mathematically, a convolutional layer cross-correlates the input and kernel and adds a scalar bias to produce an output. The two parameters of a convolutional layer are the *kernel and the scalar bias*. When training models based on convolutional layers, we typically initialize the kernels randomly, just as we would with a fully connected layer and we need to update all the components of kernel and bias during the backpropagation.

3.2 STRIDE

Stride refers to the sliding of window after each convolution operation i.e. stride governs how many cells the filter should move in an input after each convolution operation. Normally, stride value is 1, but it can be larger number as well. In above examples, stride value (s) is 1. Taking the stride value 2 in above numerical problem,

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

1	-1	-1
-1	1	-1
-1	-1	1

Image and Filter

Then the output would be:

3	-1
3	1

The resulting size of the output after applying filter with stride is given, mathematically, as

$$\left\lfloor \frac{n_h + f_h + s}{s} \right\rfloor \times \left\lfloor \frac{n_w + f_w + s}{s} \right\rfloor$$

3.3 PADDING

One of the issues with convolution is the pixel around the perimeter of image are used only once for the operation. Thus, we tend to lose the pixel from the ends of an image. To avoid so, we pad an image with 0 pixel on perimeters.

We can pad 0 pixel as necessary on any side to our input image. If we apply padding two times in image i.e. $p = 2$, then image size increases by 4 times on horizontal and vertical sides i.e. new image size becomes:

$$(n_h + 2p) \times (n_w + 2p)$$

And, after applying convolution in padded image, the size of output is:

$$\left\lfloor \frac{n_h + f_h + 2p + s}{s} \right\rfloor \times \left\lfloor \frac{n_w + f_w + 2p + s}{s} \right\rfloor$$

Also, written as,

$$\left\lfloor \frac{n_h + f_h + 2p}{s} + 1 \right\rfloor \times \left\lfloor \frac{n_w + f_w + 2p}{s} + 1 \right\rfloor$$

3.3.1 Types of Convolutions

- **Same Convolution:** If the output after applying convolution operation has same size as the input it is called same convolution.

  For example, consider an image of 6×6 and we apply padding $p = 1$, and stride $s = 1$, then the size of an output after applying filter of 3×3 is

  $$\left\lfloor \frac{6 - 3 + 2 * 1 + 1}{s} \right\rfloor \times \left\lfloor \frac{6 - 3 + 2 * 1 + 1}{s} \right\rfloor = 6 \times 6$$

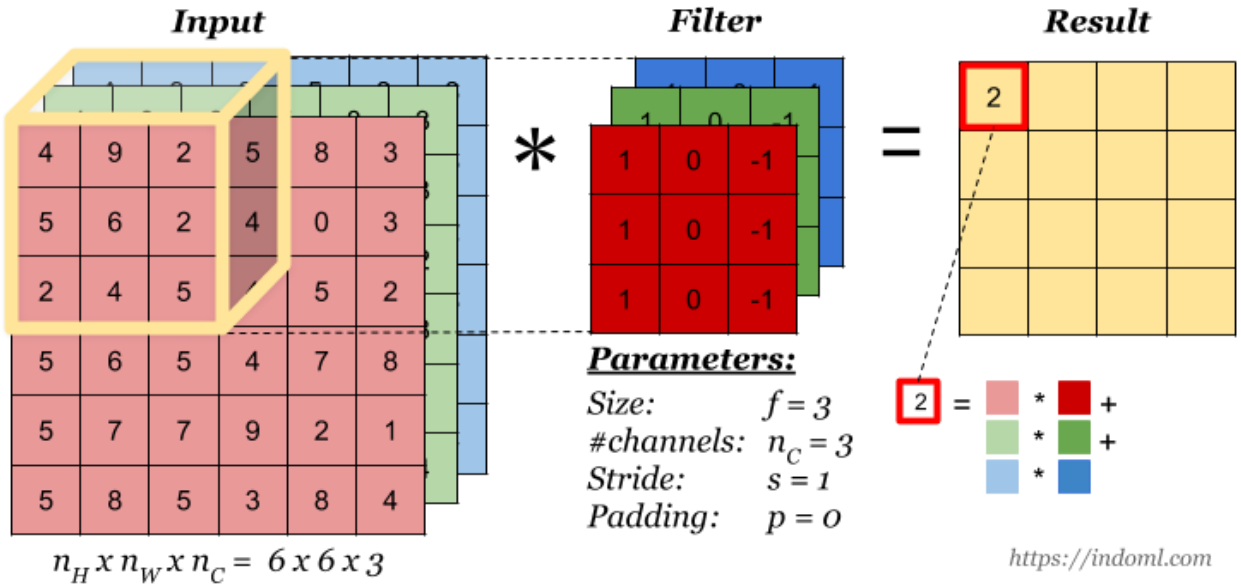
- **Valid Convolution:** If the output shrinks compared to input after applying convolution, then it is called valid convolution. We do not pad in inputs in case of valid convolution.

For example, apply no padding in above example

$$\left\lfloor \frac{6-3+2*0+1}{s} \right\rfloor \times \left\lfloor \frac{6-3+2*0+1}{s} \right\rfloor = 4 \times 4$$

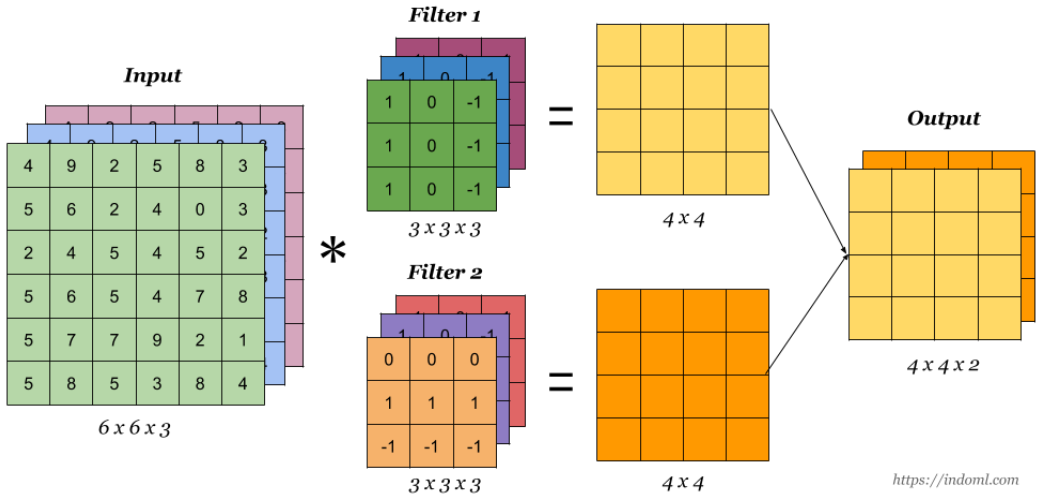
3.4 Convolution Operation on Volume

When an image has more than one channels, for example color image has 3 channels i.e. red, green and blue, the filter should also have matching number of channels.



And, to get the output, the volume of input is cross correlated with the same volume of filter and add them all.

In case of multiple filters applied, the output will have same number of channels as number of filters applied.



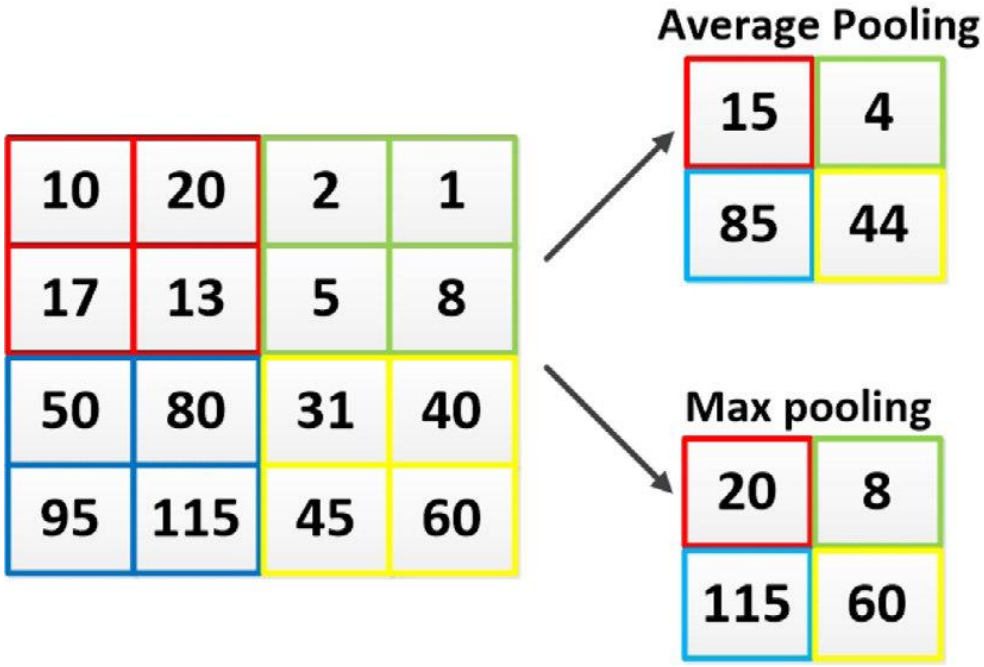
3.5 POOLING

Pooling is the act of combining two or more things. The pooling operation involves sliding pooling window over each channel of feature map and summarizing the features lying within the region covered by the filter. There are three techniques of summarization viz. maximization, averaging and minimization. This makes three types of pooling:

a) Max Pooling (*Very commonly used*)
b) Average Pooling
c) Min Pooling (Not used)

Max pooling takes the maximum value from the pooling window of input. Similarly, average pooling takes the average of all value from the pooling window of input.

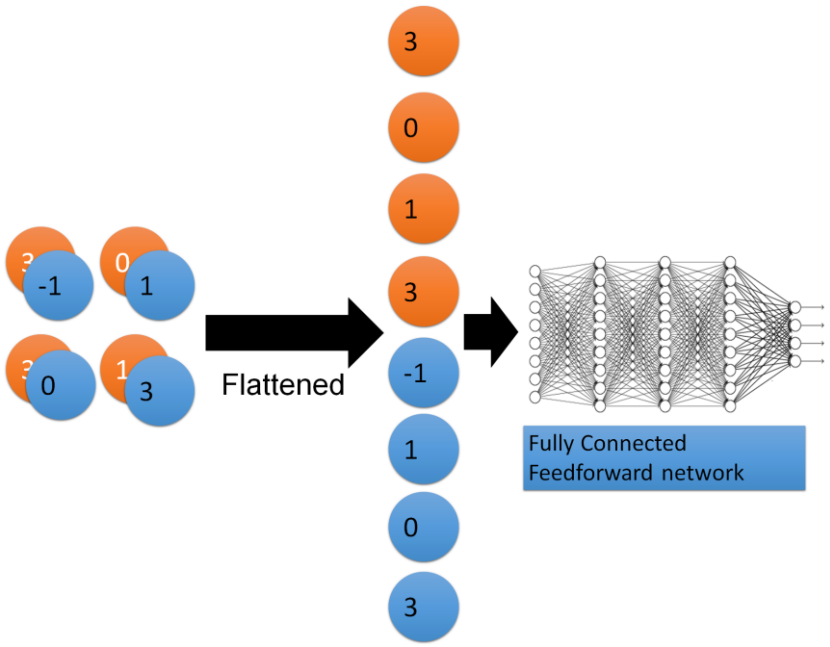
Pooling always reduces the size of input depending on the size of filter. Most commonly, pooling filter of 2×2 is used.



The reason that max pooling is common is that we are interested in those pixels which has more intensity as they contribute more to the feature in image. Pooling is also known as down sampling.

3.6 FLATTENING

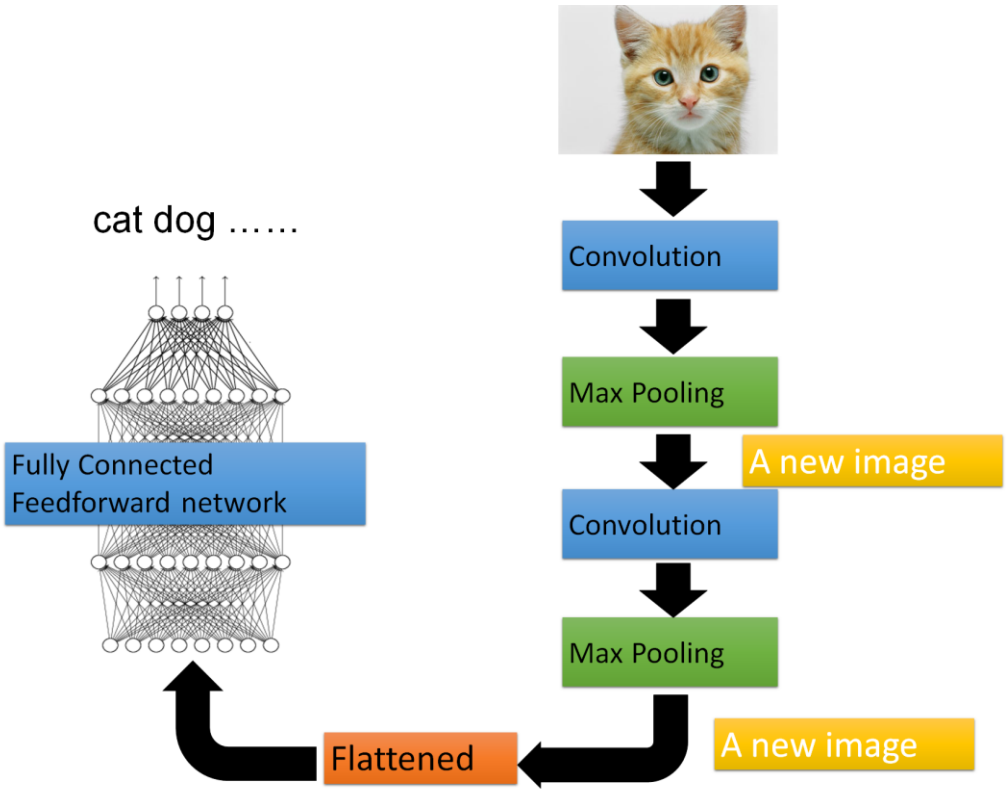
Flattening refers to the process of converting dimensional matrix to one dimensional matrix or say vector. Flattening is done before feeding the outcome of convolutional neural network or inputs like images which has matrix like structure to the fully connected neural network.



Then the flattened output is passed to the fully connected neural network.

3.7 COMPLETE CONVOLUTIONAL NEURAL NETWORK

We can now design a convolutional neural network using everything we discussed above as below:



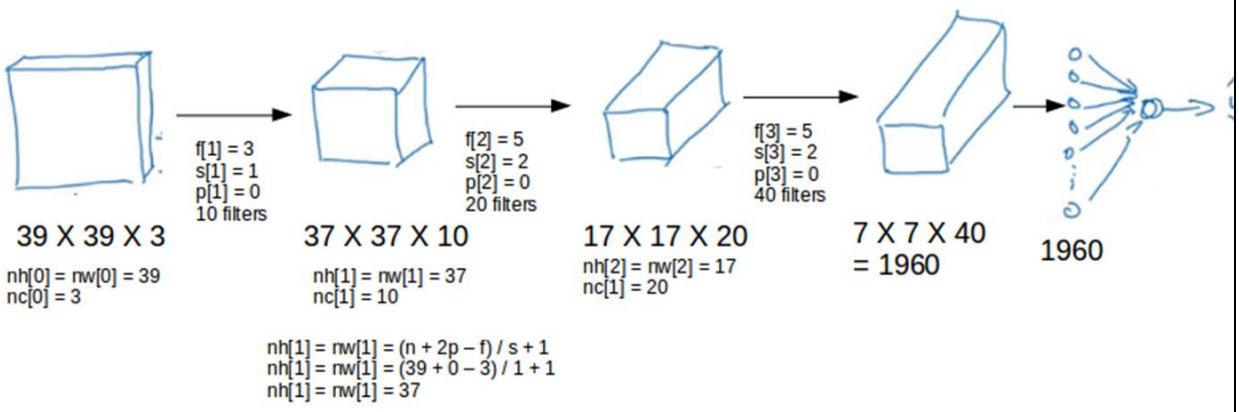
Numerical: Consider a Convolutional Neural Network having three different convolutional layers in its architecture as –
Layer-1: Filter Size – 3 X 3, Number of Filters – 10, Stride – 1, Padding – 0

Layer-2: Filter Size – 5 X 5, Number of Filters – 20, Stride – 2, Padding – 0

Layer-3: Filter Size – 5 X5 , Number of Filters – 40, Stride – 2, Padding – 0

If we give the input a 3-D image to the network of dimension 39 X 39, then determine the dimension of the vector after passing through a fully connected layer in the architecture.

Solution:

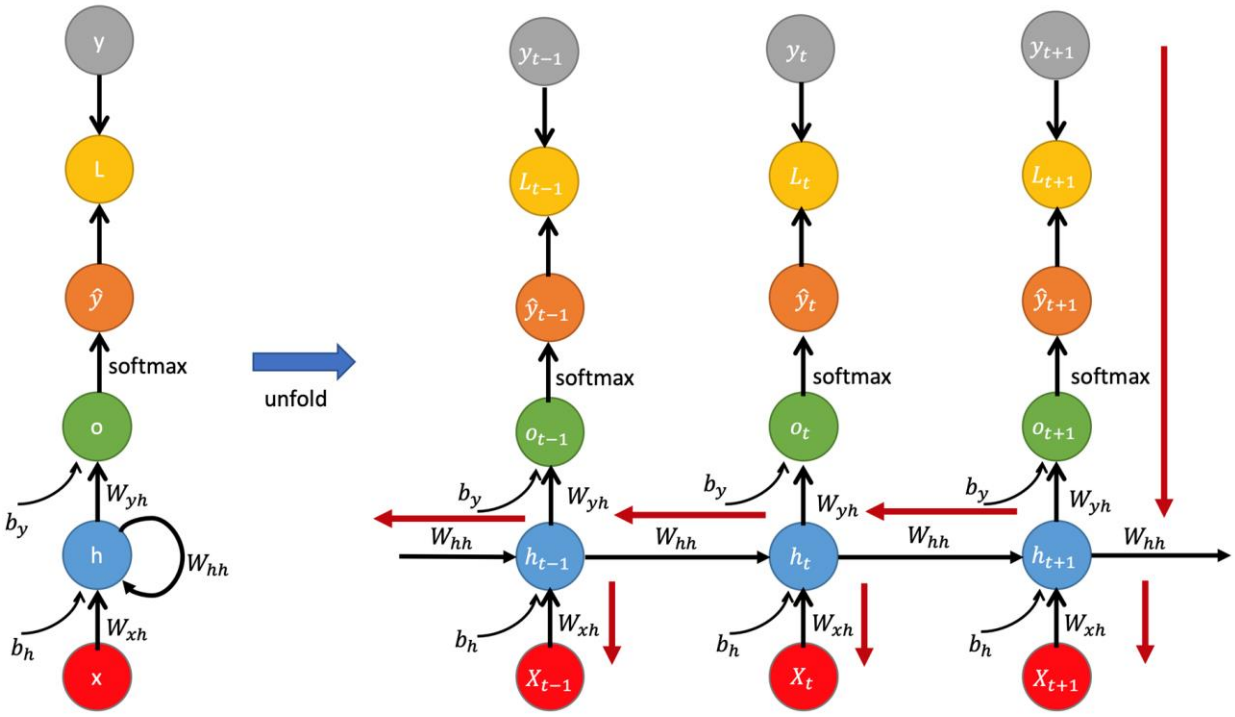


4 RECURRENT NEURAL NETWORK

A recurrent neural network (RNN) got its name from the architecture where neuron recurs exhibiting similar pattern. It is a type of artificial neural network which uses data that occurs in sequence.

These deep learning algorithms are commonly used for ordinal or temporal problems, such as language translation, natural language processing (nlp), speech recognition, and image captioning; they are incorporated into popular applications such as Siri, voice search, and Google Translate.

Like feedforward and convolutional neural networks (CNNs), recurrent neural networks utilize training data to learn. They are distinguished by their "memory" as they take information from prior inputs to influence the current input and output.



The above figure illustrates the compressed architecture and unfolded architecture of recurrent neural network. There are only two layers in the above network represented by h and o denoting hidden layer and output layer respectively. This much of layers should be enough to understand the working mechanisms of RNN. And, remember, as in fully connected neural, not all the neurons of preceding layer is connected to all the neurons of succeeding layer.

4.1 FORWARD PROPAGATION

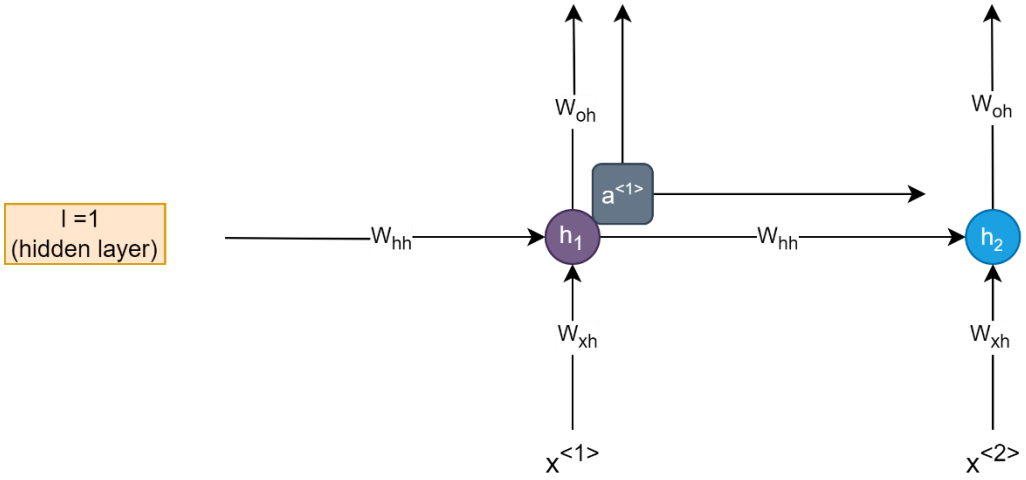
Let's start understanding the forward propagation in RNN. First input is the X_1 in the sequence which is the input to h_1 in the network with the weight of W_{xh}. Also the hidden layer h_1 gets some activation value from the previous sequence as an input which doesn't actually exist since this is the first one in sequence, but still we take some random activation h_0 with the weight of W_{hh}.

Then, in input layer l = 1, operation occurring in neuron h_1 during forward propagation can be written as:

$$z^{<1>} = W_{xh}.X_1 + W_{hh}.a^{<0>} + b^{<1>}$$

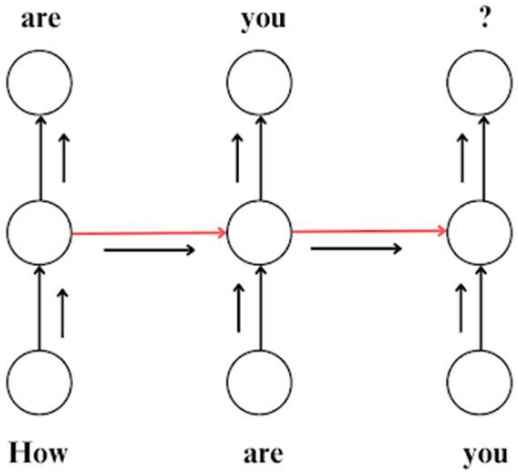
$$a^{<1>} = g(z^{<1>})$$

The output of first hidden unit from first layer $l = 1$ goes to first hidden neuron of second layer $l = 2$ and second hidden unit of first layer $l = 1$



Then, the h_2 hidden unit in first layer receives input $a^{<1>}$ from h_1 hidden unit of first layer and $x^{<2>}$ input as the second input of the input sequence.

If you are still not sure, what the sequence is then consider a sentence in English "How are you?" where the occurrence of are is dependent on How and occurrence of you on How and are. This is the sequential problem and the word occurring later has dependent over the all the already occurred words.



Now, after the completion of computation in first unit of first layer, we can proceed for the computation in the second unit of first layer,

$$z^{<2>} = W_{xh}.X_2 + W_{hh}.a^{<1>} + b^{<2>}$$

$$a^{<2>} = g(z^{<2>})$$

And so on.

Thus, the forward propagation in RNN, can be mathematically written as:

In hidden layer $l = 1$, for $t = 1 \dots T$

$$z^{[1]<t>} = W_{xh}^{[1]}.a_t^{[1]} + W_{hh}^{[1]}.a^{<t-1>} + b^{[1]<t>}$$

$$a^{<t>} = g_h^{[1]}\left(z^{[1]<t>}\right)$$

Take a time to digest this.

Here,

W_{xh} *is the weight of the inputs to the hidden unit* ,

W_{hh} *is the weight of the inputs from and to the hidden unit*, and,

Wyh *is the weight of the output from the hidden unit when passing to another layer*

Now, in second layer as in first layer, forward propagation computation happens

In output layer $l = 2$, for $t = 1 \dots T$

$$z^{[2]<t>} = W_{oh}^{[1]}.a^{[1]<t>} + W_{hh}^{[2]}.a^{[1]<t-1>} + b^{[2]<t>}$$

$$a^{[2]<t>} = g_h^{[2]}\left(z^{[2]<t>}\right)$$

And so on for all the other layers, if they exist.

Forward Propagation in RNN

$for\ l = 1 \dots L-1$

for $t = 1 \dots T$

$$z^{[l]<t>} = W_{xh}^{[l]}.a^{[t]} + W_{hh}^{[l]}.a^{<t-1>} + b^{[l]<t>}$$

$$a^{[l]<t>} = g_h^{[1]}\left(z^{[l]<t>}\right)$$

Here, $W_{oh}^{[l-1]} = W_{xh}^{[l]}$

$for\ l = L$

for $t = 1 \dots T$

$$z^{[L]<t>} = W_{xh}^{[L]} . a^{[L]<t>} + W_{hh}^{[l]} . a^{<t-1>} + b^{[l]<t>}$$

$$a^{[l]<t>} = g_h^{[1]}(z^{[l]<t>})$$

4.2 Back Propagation Through Time

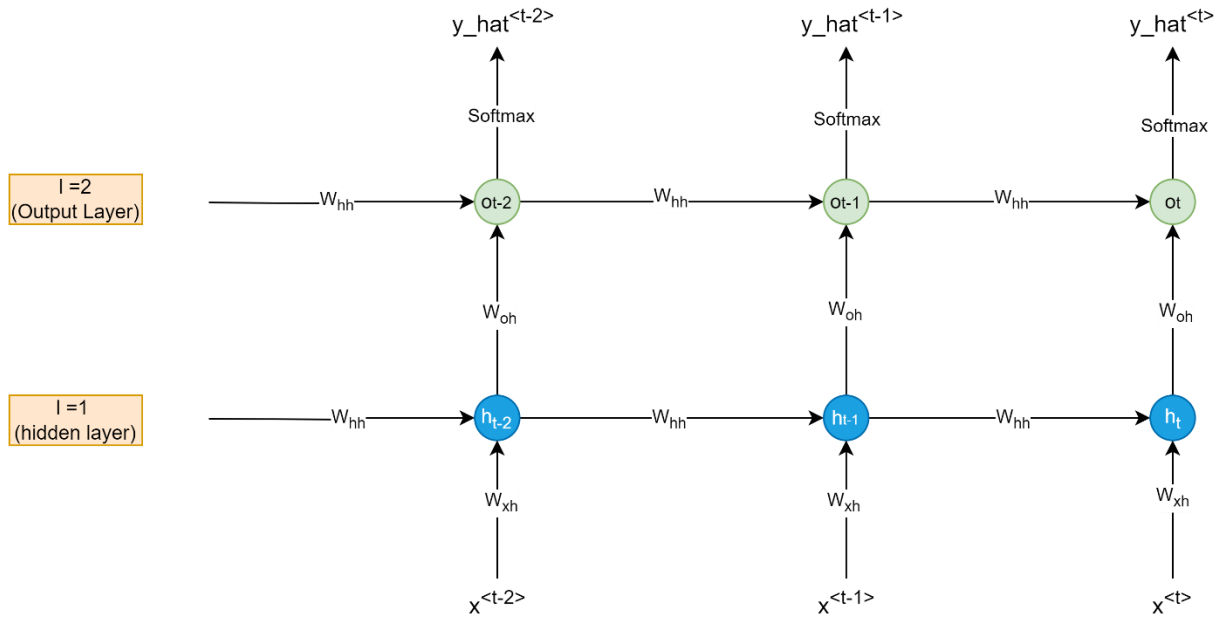
Recurrent Neural Networks specializes in processing the sequences of data, introducing the concept of time steps, where each step corresponds to a specific moment in the sequence. This temporal aspect enables RNNs to excel in tasks involving sequential data like text, speech, and time series.

A time step in a Recurrent Neural Network (RNN) refers to a specific moment or instance in a sequence of data being processed by the network. In the context of sequential data, such as text, speech, or time series, the RNN processes one element of the sequence at each time step.

For RNNs to learn a pattern from the sequential data, a variant of the backpropagation algorithm known as "**Backpropagation Through Time**" (BPTT) is used.

Let's again understand the back propagation through time step by step:

Consider we have only one-time step sequence in the network, then this will be a simple neural network where there is one neuron in each layer.



Consider C be the Cost function used to calculate error irrespective of classification or regression function. Then, as a first step in the back propagation we need to compute,

$$\frac{\partial C}{\partial \hat{y}}$$

And, as per the notation introduced above, in the back propagation, we can write,

$$d\hat{y} = \frac{\partial C}{\partial \hat{y}}$$

And, next, we compute derivative of Cost with respect to W_{oh},

$$dW_{oh} = \frac{\partial C}{\partial W_{oh}} = \frac{\partial C}{d\hat{y}} . \frac{\partial \hat{y}}{dW_{oh}}$$

Next, we compute derivative of Cost with respect to a

$$da = \frac{\partial C}{\partial a} = \frac{\partial C}{\partial \hat{y}} . \frac{\partial \hat{y}}{\partial a}$$

Similarly, we compute the derivative of Cost with respect to W_{hh}

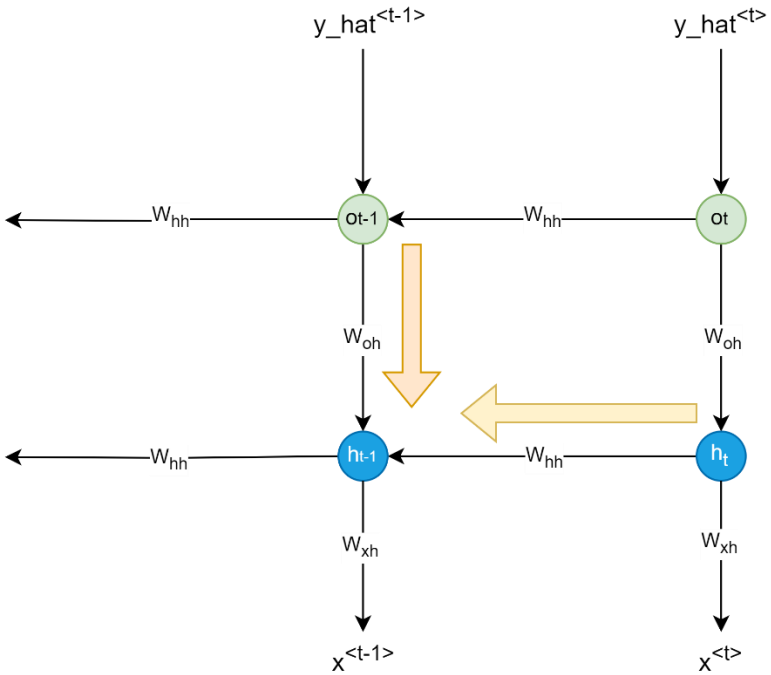
$$dW_{hh} = \frac{\partial C}{d\hat{y}} . \frac{\partial \hat{y}}{\partial a} . \frac{\partial a}{dW_{hh}}$$

And, lastly, we compute the derivative of Cost with respect to W_{xh}

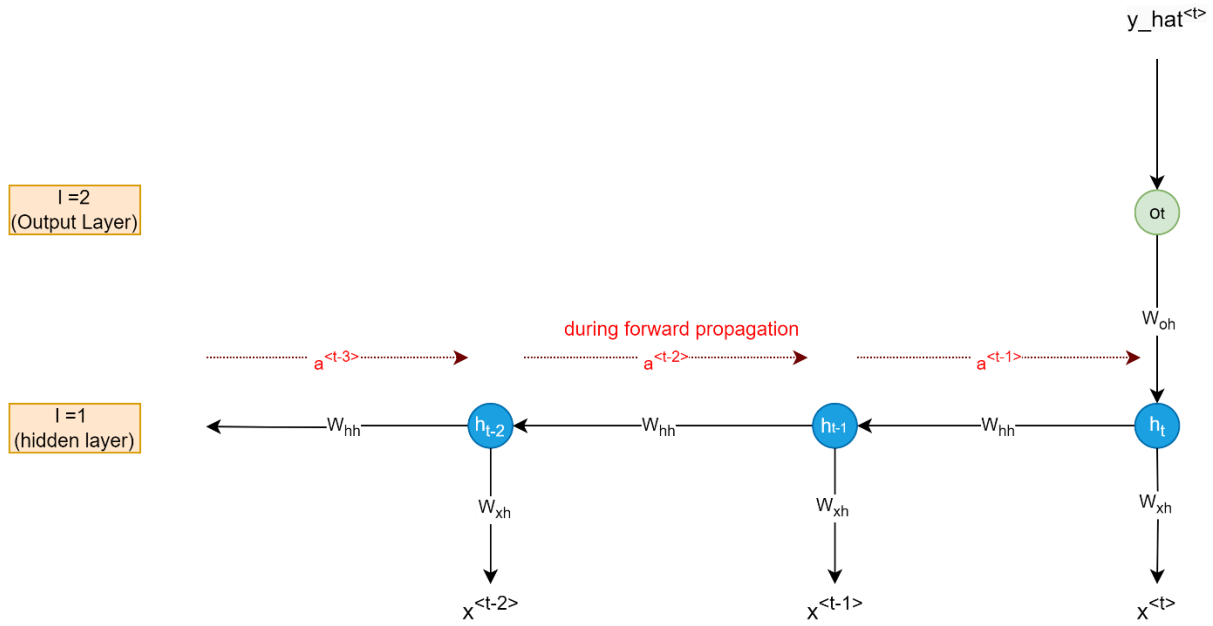
$$dW_{xh} = \frac{\partial C}{d\hat{y}} . \frac{\partial \hat{y}}{\partial a} . \frac{\partial a}{dW_{xh}}$$

<u>For time step $t-1$ based on time step t</u>

Then, for time step $t-1$, the loss back propagates from t-th sequence and from $t-1$ sequence itself. Now, we have a situation here. Let's be very clear that at time step t-1, we are receiving two version of loss back propagated i.e. $da^{<t-1>}$. Now, both the version of losses is used to make update on other parameters following $a^{<t-1>}$ one by one.



Let's first understand the back propagation from t-th sequence. During the forward propagation W_{hh} from $t-1$ time step propagates forward to t-th time step along with the activation from $t-1$ time step i.e. $a^{<t-1>}$ (*check the forward propagation once again to confirm*).



Thus, during the back propagation, time step $t-1$ receives loss from t-th step. So, in that case, we compute

$$da^{<t-1>} = \frac{\partial C}{\partial a^{<t-1>}} = \frac{\partial C}{\partial \hat{y}^{<t>}} \cdot \frac{\partial \hat{y}^{<t>}}{\partial a^{<t>}} \cdot \frac{\partial a^{<t>}}{\partial a^{<t-1>}}$$

Because, $a^{<t>}$ is now the function of $a^{<t-1>}$(*shown in above figure*).

Remember from the forward propagation, we have

for $t = 1 \ldots T$

$$z^{[1]<t>} = W_{xh}^{[1]} . a_t^{[1]} + W_{hh}^{[1]} . a^{<t-1>} + b^{[1]<t>}$$

$$a^{<t>} = g_h^{[1]}(z^{[1]<t>})$$

i.e.

$$a^{<t>} = g_h^{[1]}\left(W_{xh}^{[1]} . a^{[1]<t>} + W_{hh}^{[1]} . a^{<t-1>} + b^{[1]<t>}\right)$$

Which means, $a^{<t>}$ is the function of $a^{<t-1>}$.

Now, from the forward propagation, we know

$$a^{<t-1>} = g_h^{[1]}\left(W_{xh}^{[1]} . a^{[0]<t-1>} + W_{hh}^{[1]} . a^{<t-2>} + b^{[1]<t-1>}\right)$$

So, here

$a^{[0]<t-1>}$ is the $t-1$ input of the sequence. For example, if our input string is "Hello Sir", "Hello" is the $(t-1)$-st input and "Sir" is the t-th input.

Then, it is obvious to determine, we can compute the next derivative w.r.t. W_{xh} at time step $t-1$.

**Note: The derivative we computed for W_{xh} at time step t doesn't work here. We again have to compute.*

$$dW_{xh} = \frac{\partial C}{\partial W_{xh}} = \frac{\partial C}{\partial \hat{y}^{<t>}} \cdot \frac{\partial \hat{y}^{<t>}}{\partial a^{<t>}} \cdot \frac{\partial a^{<t>}}{\partial a^{<t-1>}} \cdot \frac{\partial a^{<t-1>}}{\partial W_{xh}}$$

But, we can accumulate the derivative W_{xh} from all the sequence and write as:

$$dW_{xh} = \frac{\partial C}{\partial W_{xh}} = \frac{\partial C}{\partial \hat{y}^{<t>}} \cdot \frac{\partial \hat{y}^{<t>}}{\partial a^{<t>}} \cdot \frac{\partial a^{<t>}}{\partial a^{<t-1>}} \cdot \frac{\partial a^{<t-1>}}{\partial W_{xh}} + \frac{\partial C}{d\hat{y}^{<t>}} \cdot \frac{\partial \hat{y}^{<t>}}{\partial a^{<t>}} \cdot \frac{\partial a^{<t>}}{\partial W_{xh}}$$

Now, similarly, if we consider three-time steps, we get

$$dW_{xh} = \frac{\partial C}{\partial W_{xh}} = \frac{\partial C}{\partial \hat{y}^{<t>}} \cdot \frac{\partial \hat{y}^{<t>}}{\partial a^{<t>}} \cdot \frac{\partial a^{<t>}}{\partial a^{<t-1>}} \cdot \frac{\partial a^{<t-1>}}{\partial a^{<t-2>}} \cdot \frac{\partial a^{<t-2>}}{\partial W_{xh}} + \frac{\partial C}{\partial \hat{y}^{<t>}} \cdot \frac{\partial \hat{y}^{<t>}}{\partial a^{<t>}} \cdot \frac{\partial a^{<t>}}{\partial a^{<t-1>}} \cdot \frac{\partial a^{<t-1>}}{\partial W_{xh}} + \frac{\partial C}{d\hat{y}^{<t>}} \cdot \frac{\partial \hat{y}^{<t>}}{\partial a^{<t>}} \cdot \frac{\partial a^{<t>}}{\partial W_{xh}}$$

Backward Propagation through time

Thus, $for\ t = 1 \ldots T$, from above we can write for W_{xh},

$$dW_{xh} = \frac{\partial C}{\partial W_{xh}} = \sum_{t=0}^{T} \frac{\partial C}{\partial \hat{y}^{<T-t>}} \cdot \frac{\partial \hat{y}^{<T-t>}}{\partial h^{<T>}} \left(\prod_{j=T-t+1}^{T} \frac{\partial h^{<T-j+1>}}{\partial h^{T-j}} \right) \frac{\partial h^{<T-t-1>}}{\partial W_{xh}}$$

Similarly, for W_{hh}

$$dW_{hh} = \frac{\partial C}{\partial W_{hh}} = \sum_{t=0}^{T} \frac{\partial C}{\partial \hat{y}^{<T-t>}} \cdot \frac{\partial \hat{y}^{<T-t>}}{\partial h^{<T>}} \left(\prod_{j=T-t+1}^{T} \frac{\partial h^{<T-j+1>}}{\partial h^{T-j}} \right) \frac{\partial h^{<T-t-1>}}{\partial W_{hh}}$$

$$dW_{oh} = \frac{\partial C}{\partial W_{oh}} = \sum_{t=0}^{T} \frac{\partial C}{\partial \hat{y}^{<T-t>}} \cdot \frac{\partial \hat{y}^{<T-t>}}{\partial h^{<T>}} \left(\prod_{j=T-t+1}^{T} \frac{\partial h^{<T-j+1>}}{\partial h^{T-j}} \right) \frac{\partial h^{<T-t-1>}}{\partial W_{oh}}$$

Similarly, Taking bias in each neuron, for b_h,

$$db_h^{<t>} = \sum_{t=0}^{T} \frac{\partial C}{\partial \hat{y}^{<T-t>}} \cdot \frac{\partial \hat{y}^{<T-t>}}{\partial h^{<T-t>}} \frac{\partial h^{<T-t>}}{\partial b_h}$$

And b_o,

$$db_o^{<t>} = \sum_{t=0}^{T} \frac{\partial C}{\partial \hat{y}^{<T-t>}} \cdot \frac{\partial \hat{y}^{<T-t>}}{\partial b_o}$$

And so on for the output of time sequence $t-1, t-2, \ldots, t=0$.

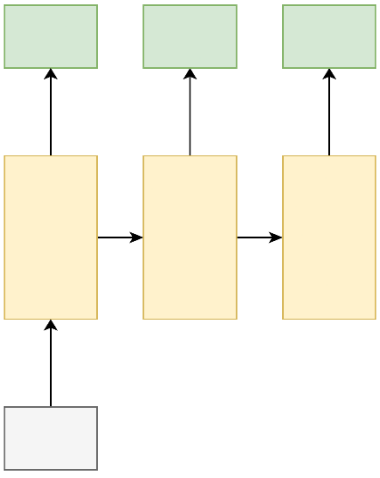
4.3 TYPES OF RNN

Based on inputs and outputs there are following four types of RNN.

1. One to One
   RNN with one input and one output

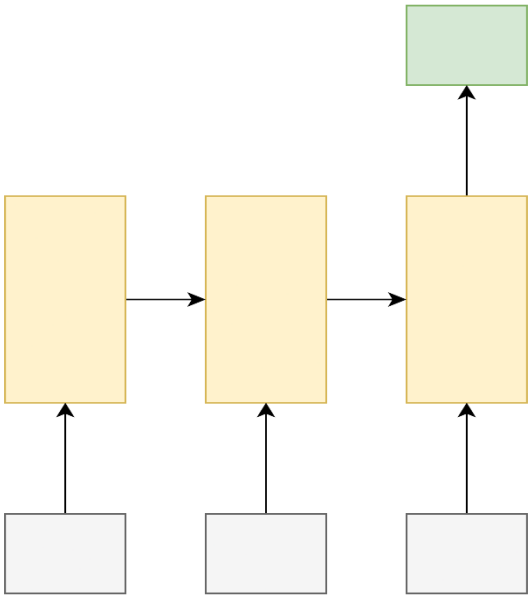
   E.g.

2. One to Many
   RNN with one input but many output



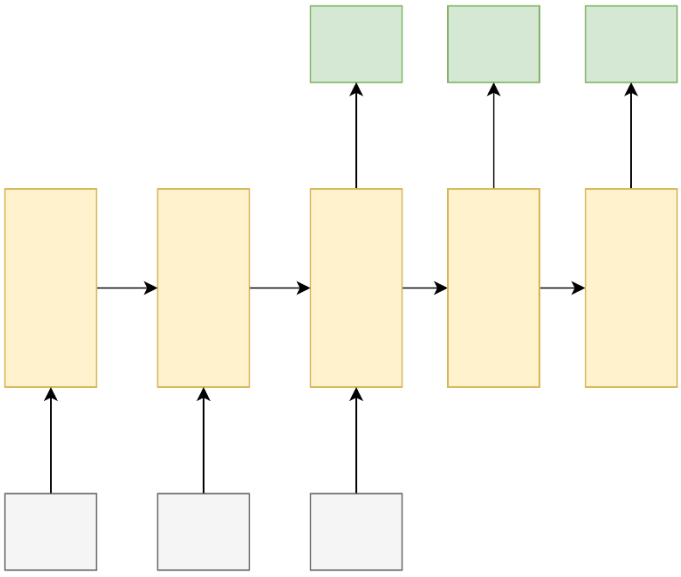
   E.g. Text Generation, Music Generation etc.

3. Many to One
   RNN with many inputs but one output



E.g. Sentiment Classification etc.

4. Many to Many
   RNN with Many inputs and many output



Eg. Language Translation, Text Summarization

4.4 VANISHING/EXPLODING GRADIENT

The vanishing and exploding gradient phenomena are often encountered in the context of RNNs. The reason why they happen is that it is difficult to capture long term dependencies because of multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers.

For example,

If you check on the formula we derived above,

$$dW_{xh} = \frac{\partial C}{\partial W_{xh}} = \sum_{t=0}^{T} \frac{\partial C}{\partial \hat{y}^{<T-t>}} \cdot \frac{\partial \hat{y}^{<T-t>}}{\partial h^{<T>}} \left(\prod_{j=T-t+1}^{T} \frac{\partial h^{<T-j+1>}}{\partial h^{T-j}} \right) \frac{\partial h^{<T-t-1>}}{\partial W_{xh}}$$

We can see the product of derivative involved.

If the derivative value happens to be very small than 1 but greater than 0 (e.g. 0.025) and multiplication with similar next derivate value (e.g. 0.1) will result to smaller value. If this is the case, then during the back propagation from T-th sequence to 1st sequence, will have so many multiplications that the ultimate product value may result to 0 which means the derivative value of cost (C) against weight (W) will be 0 which means no update to be applied on update rule.

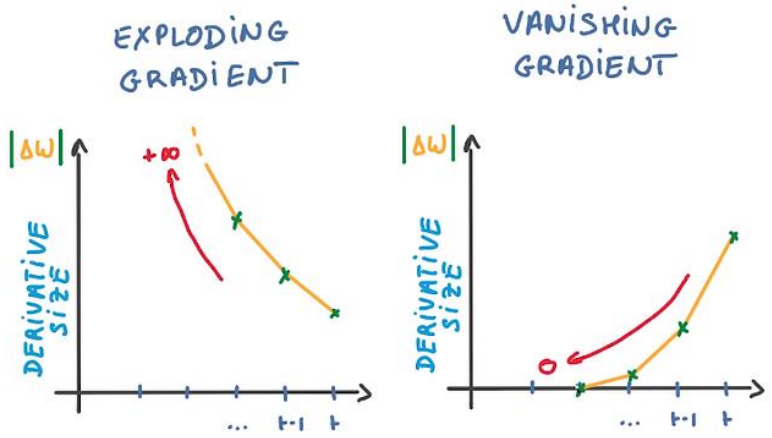
v

$$w = w - \alpha * dC = w \; [\because dC = 0]$$

This is called the **vanishing gradient**. This is also known as the decay of information over time.

Similarly, if the value of derivative is greater than 1, in fact very larger than 1, the resulting multiplication value will be too large and the weight update will be largely influenced by large value of update. This is known as **exploding gradient**.

As Sigmoid function results to 0 and 1 output, this often leads to vanishing gradient. Thus, it is not recommended to use with RNN. Instead, tangent function is used with RNN which results between 1 and -1.



4.5 DROPOUT

Dropout is the concept of temporarily dropping or replacing random neurons from hidden layer of network. Dropping the neuron will disable the to and from path of neuron. Dropout is performed only during the training phase but not during the testing and prediction. The reason is, dropout is performed in the network to avoid overfitting (memorizing the pattern from data by specific neuron) during training phase. During the dropout, random number of neurons are dropped usually 10 or 20% of neurons. Thus, dropout is also known as a type of regularization in deep learning.