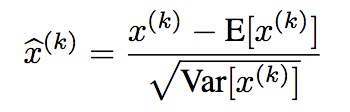
**What is batch normalization?**

First, it is important to note that in a neural network, things will go well if your input to the network is mean subtracted. In addition, sometimes they also normalize the input data and make the standard deviation equal to 1 in addition to mean subtraction.

Now, since every layer of the neural network accepts input from the previous layer, it makes sense to normalize the data flowing into any particular layer of the network.

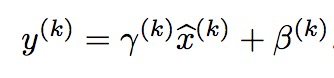
Batch Normalization is that small component that sits between the layers of the neural network, continuously taking the output of a particular layer and normalizing it before sending it across to the next layer as input.



basically is taking the output of layer (k) and normalizing it. Remember that the output of layer (k) itself is a mini batch of N samples. Each sample will have D features. So you have a matrix of N X D and so you take every column of this matrix and subtract the mean and divide by standard deviation to normalize the data. This then goes ahead to the next layer and the process if repeated at the output of the next layer, and so on.

But note, that batch normalization may not always be good for the network. So, we need to provide the network the ability to to UNDO and recover back the original output of the current layer even after batch normalization just in case it thinks (or learns) that batch norm is not a good thing to do at a particular layer.

So, the actual batch normalization component actually has the equation as



where gamma and beta provide the ability for the batch norm component to recover back the original output of the current layer. As you can guess, if gamma and beta are learned by the network to the following values,

https://qph.fs.quoracdn.net/main-qimg-8dd6e4dac30ffaee9eb09ffaea6b542d

https://qph.fs.quoracdn.net/main-qimg-aa0870411e58afa732b4219c58258788

**Why does batch normalization work well?**

1. It basically normalizes the data and normalized data is almost always good before input to any layer.
2. It acts as a regularizer. This is because the input to the next layer for a training sample is calculated based on the combined operations of all training samples in this particular mini-batch that is batch normed. So there is some regularization here.
3. It allows you to use higher learning rates and so networks with batch normalization are usually faster than those that do not have batch normalization.

<https://stackoverflow.com/questions/40708169/how-to-initialize-biases-in-a-keras-model>

* [**What is the difference between back-propagation and feed-forward Neural Network?**](https://stackoverflow.com/questions/28403782/what-is-the-difference-between-back-propagation-and-feed-forward-neural-network)A [Feed-Forward Neural Network](http://en.wikipedia.org/wiki/Feedforward_neural_network) is a type of Neural Network **architecture** where the connections are "fed forward", i.e. do not form cycles (like in recurrent nets).
* The term "Feed forward" is also used when you input something at the input layer and it travelsfrom input to hidden and from hidden to output layer.   
  **The values are "fed forward"**.

Both of these uses of the phrase "feed forward" are in a context that has nothing to do with training per se.

* Backpropagation is a **training algorithm** consisting of 2 steps: 1) **Feed forward** the values 2) calculate the error and **propagate it back** to the earlier layers. So to be precise, forward-propagation is part of the backpropagation **algorithm** but comes before back-propagating.

<https://stackoverflow.com/questions/28403782/what-is-the-difference-between-back-propagation-and-feed-forward-neural-network>

**What is learning rate?**

In simple words learning rate determines how fast weights (in case of a neural network) or the co-efficient (in case of linear regression or logistic regression) change.  
If c is a cost function with variables (or weights) w1,w2….wn then,  
Let’s take stochastic gradient descent where we change weights sample by sample -

**For every sample(  
w1\_new= w1 + (learning rate)\* (derivative of cost function wrt w1)  
)**

If learning rate is too high derivative may miss the 0 slope point or learning rate is too low then it may take forever to reach that point. So we need to figure out that balanced learning rate.

Now another thing is learning rate may not be a constant for all the layers of a neural network, it may be different for different layers which avoids problem of vanishing gradient i.e, weights may stop changing as weight change backpropogates itself to first layer (since there are lot multiplications of derivatives and these derivatives itself REACH decimal values < 1 and there products are even smaller if we observe mathematical analysis of backpropagation of neural networks and as a result learning will not take place and saturate immaturely) so we assign variable learning rate to each layer.

**How to assign variable learning rate to each layer?**

[How to set layer-wise learning rate in Tensorflow?](https://stackoverflow.com/questions/34945554/how-to-set-layer-wise-learning-rate-in-tensorflow)

It can be achieved quite easily with 2 optimizers:

var\_list1 = [variables from first 5 layers]

var\_list2 = [the rest of variables]

train\_op1 = GradientDescentOptimizer(0.00001).minimize(loss, var\_list=var\_list1)

train\_op2 = GradientDescentOptimizer(0.0001).minimize(loss, var\_list=var\_list2)

train\_op = tf.group(train\_op1, train\_op2)

One disadvantage of this implementation is that it computes tf.gradients(.) twice inside the optimizers and thus it might not be optimal in terms of execution speed. This can be mitigated by explicitly calling tf.gradients(.), splitting the list into 2 and passing corresponding gradients to both optimizers.

more efficient but longer implementation:

var\_list1 = [variables from first 5 layers]

var\_list2 = [the rest of variables]

opt1 = tf.train.GradientDescentOptimizer(0.00001)

opt2 = tf.train.GradientDescentOptimizer(0.0001)

grads = tf.gradients(loss, var\_list1 + var\_list2)

grads1 = grads[:len(var\_list1)]

grads2 = grads[len(var\_list1):]

tran\_op1 = opt1.apply\_gradients(zip(grads1, var\_list1))

train\_op2 = opt2.apply\_gradients(zip(grads2, var\_list2))

train\_op = tf.group(train\_op1, train\_op2)

You can use tf.trainable\_variables() to get all training variables and decide to select from them. The difference is that in the first implementation tf.gradients(.) is called twice inside the optimizers. This may cause some redundant operations to be executed (e.g. gradients on the first layer can reuse some computations for the gradients of the following layers).

[Keras resume training with different learning rate](https://stackoverflow.com/questions/47994638/keras-resume-training-with-different-learning-rate)?

**How to implement batch normalization in Keras?**

**What is optimal batch size for Neural Net?**

[**Why mini batch size is better than one single “batch” with all training data?**](https://datascience.stackexchange.com/questions/16807/why-mini-batch-size-is-better-than-one-single-batch-with-all-training-data)

The key advantage of using minibatch as opposed to the full dataset goes back to the fundamental idea of stochastic gradient descent[1](https://arxiv.org/abs/1804.07612).

In batch gradient descent, you compute the gradient over the entire dataset, averaging over potentially a vast amount of information. It takes lots of memory to do that. But the real handicap is the batch gradient trajectory land you in a bad spot (saddle point).

In pure SGD, on the other hand, you update your parameters by adding (minus sign) the gradient computed on a *single* instance of the dataset. Since it's based on one random data point, it's very noisy and may go off in a direction far from the batch gradient. However, the noisiness is exactly what you want in non-convex optimization, because it helps you escape from saddle points or local minima (Theorem 6 in [2]). The disadvantage is it's terribly inefficient and you need to loop over the entire dataset many times to find a good solution.

The minibatch methodology is a compromise that injects enough noise to each gradient update, while achieving a relative speedy convergence.

**ANN vs RNN?**

**RNN vs LSTM?**

**What happens if we input all data to neural network?**

**What happens if we input single row to neural network?**

**What is Bias Term, how does it work? When do we use Bias?**

[**What is bias in artificial neural network?**](https://www.quora.com/What-is-bias-in-artificial-neural-network/)

**What is difference between ANN and CNN?**

The conventional-neural-networks is a subclass of neural-networks which have at least one convolution layer. They are great for capturing local information (e.g. neighbor pixels in an image or surrounding words in a text) as well as reducing the complexity of the model (faster training, needs fewer samples, reduces the chance of overfitting).

The main difference is that the convolutional neural network (CNN) has layers of convolution and pooling.

Convolutional layers take advtage of the local spatial coherence of the input. This is only possible because we assume that spatially close inputs are correlated. For images, this can be seen by the fact that the image loses its meaning when the pixels are shuffled.

Using this property, CNNs are able to cut down on the number of parameter by sharing weights. This makes them extremely efficient in image processing, compared to multi-layer perceptron.

**Sigmoid Function Vs Logistic Regression?**

The sigmoid function is a special case of the Logistic function when L=1, k=1, x0=0.

If you [play around with the parameters (Wolfram Alpha)](http://www.wolframalpha.com/input/?i=plot+2%2F%7B1%2Bexp%7B-2%7Bx-0%7D%7D%7D,+with+-10+%3C+x+%3C+10), you will see that

* L is the maximum value the function can take. e−k(x−x0) is always greater or equal than 0, so the maximum point is achieved when it it 0, and is at L/1.
* x0 controls where on the x axis the growth should the, because if you put x0 in the function, x0-x0 cancel out and e0=1, so you end up with f(x0) =L/2, the midpoint of the growth.
* the parameter k controls how steep the change from the minimum to the maximum value is.

[**What is the purpose of an activation function in Neural Networks?**](https://ai.stackexchange.com/questions/5493/what-is-the-purpose-of-an-activation-function-in-neural-networks)

It is said that activation functions in neural networks help introduce ***non-linearity***.

* What does this mean?
* What does ***non-linearity*** mean in this context?
* How does introduction of this ***non-linearity*** help?
* Are there any other purposes of ***activation functions***?

First, what does non-linearity mean? It means something (a function in this case) which is not linear with respect to a given variable/variables. i.e. f(c1.x1+c2.x2...cn.xn+b)!=c1.f(x1)+c2.f(x2)...cn.f(xn)+b.f(c1.x1+c2.x2...cn.xn+b)!=c1.f(x1)+c2.f(x2)...cn.f(xn)+b. `

What does non-linearity mean in this context? It means that the Neural Network can successfully approximate functions (up-to a certain error ee decided by the user) which does not follow linearity or it can successfully predict the class of a function which is divided by a decision boundary which is not linear.

Why does it help? I hardly think you can find any physical world phenomenon which follows linearity straightforwardly. So you need a non-linear function which can approximate the non-linear phenomenon. Also a good intuition would be any decision boundary or a function is a linear combination of polynomial combinations of the input features (so ultimately non-linear).

Purposes of activation function? In addition to introducing non-linearity every activation function has its own features.

**What are different activation functions and their uses?**

**Sigmoid**1/(1+e−(w1∗x1...wn∗xn+b))

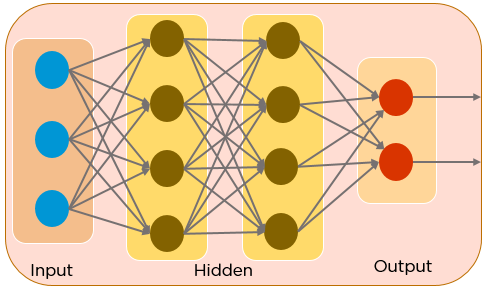
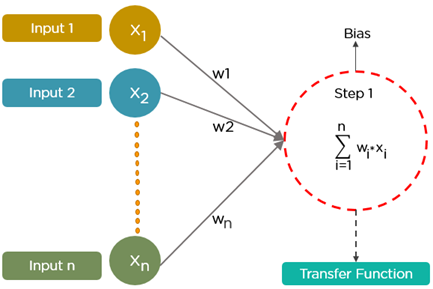
This is one of the most common activation function and is monotonically increasing everywhere. This is generally used at the final output node as it squashes values between 0 and 1 (if output is required to be 0 or 1).Thus above 0.5 is considered 1 while below 0.5 as 0, although a different threshold (not 0.5) maybe set. Its main advantage is that its differentiation is easy and uses already calculated values and supposedly horseshoe crab neurons have this activation function in their neurons.

**Tanh** e(w1∗x1...wn∗xn+b)−e−(w1∗x1...wn∗xn+b))/

(e(w1∗x1...wn∗xn+b)+e−(w1∗x1...wn∗xn+b)

This has an advantage over the sigmoid activation function as it tends to centre the output to 0 which has an effect of better learning on the subsequent layers (acts as a feature normaliser). A nice explanation [here](https://stats.stackexchange.com/questions/101560/tanh-activation-function-vs-sigmoid-activation-function). Negative and positive output values maybe considered as 0 and 1 respectively. Used mostly in RNN's.

**How do Neural Networks work?**

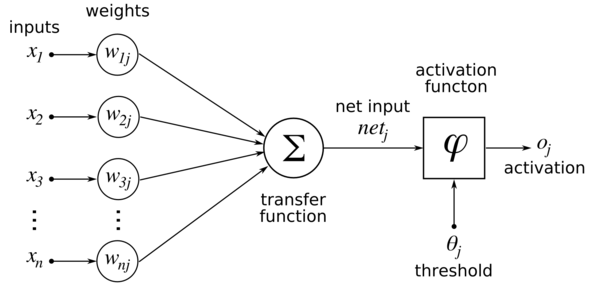
 

An artificial neural network consists of several **parameters** and **hyperparameters** that drive the output of a neural network model. Some of these parameters are **weights, biases, number of epochs, the learning rate, batch size, number of batches**, etc.

Each node in the network has some weights assigned to it. A **transfer function** is used to calculate the weighted sum of inputs and a bias is added.

**Backpropagation** is a technique used to minimize the error in the network. This error is calculated with the help of a **cost function**. We back-propagate the error and adjust the weights randomly to minimize the error. This process is repeated several times till we make sure that the difference between the predicted and original output is least.

**What is the transfer function in Artificial Neural Networks?**  
Activation function and transfer function are the same but latter is used less often.



**What is Gradient Descent?**

Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the [parameters](https://ml-cheatsheet.readthedocs.io/en/latest/glossary.html#glossary-parameters) of our model. Parameters refer to coefficients in [Linear Regression](https://ml-cheatsheet.readthedocs.io/en/latest/linear_regression.html) and [weights](https://ml-cheatsheet.readthedocs.io/en/latest/nn_concepts.html#nn-weights) in neural networks.

* Batch Gradient Descent
* Stochastic Gradient Descent

**What is Flatten layer in Keras?**

You will see that:

Dense(16, input\_shape=(5,3))

would result in a Dense network with 3 inputs and 16 outputs which would be applied independently for each of 5 steps. So if D(x) transforms 3 dimensional vector to 16-d vector what you'll get as output from your layer would be a sequence of vectors: [D(x[0,:], D(x[1,:],..., D(x[4,:]] with shape (5, 16). In order to have the behaviour you specify you may first Flatten your input to a 15-d vector and then apply Dense:

model = Sequential()

model.add(Flatten(input\_shape=(3, 2)))

model.add(Dense(16))

model.add(Activation('relu'))

model.add(Dense(4))

model.compile(loss='mean\_squared\_error', optimizer='SGD')

**When do we use Flattening layer?**

Flattening is the process of converting all the resultant 2 dimensional arrays into a single long continuous linear vector.  
The process of building a Convolutional Neural Network always involves four major steps.

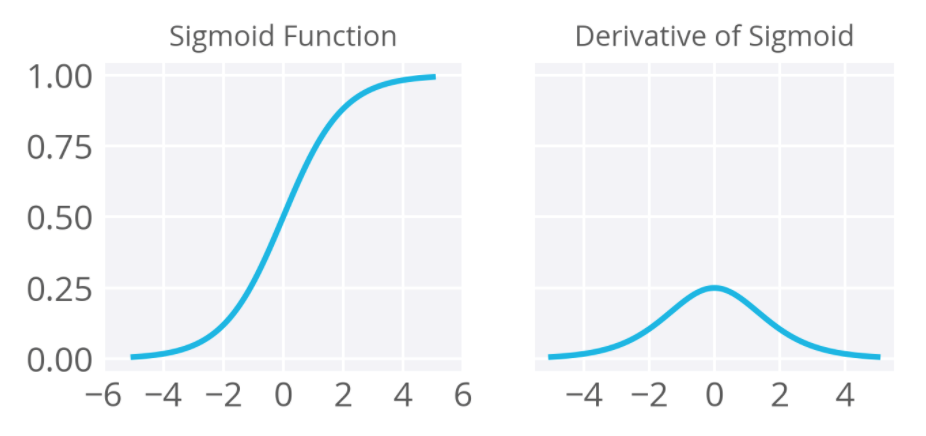
**Step - 1 : Convolution**  
**Step - 2 : Pooling**  
**Step - 3 : Flattening**  
**Step - 4 : Full connection**

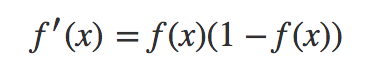
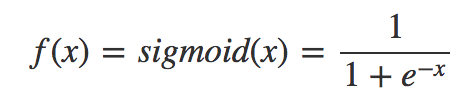
The flattening step is needed so that you can make use of fully connected layers after some convolutional layers. Fully connected layers don't have a local limitation like convolutional layers (which only observe some local part of an image by using convolutional filters). This means you can combine all the found local features of the previous convolutional layers. Each feature map channel in the output of a CNN layer is a "flattened" 2D array created by adding the results of multiple 2D kernels (one for each channel in the input layer).

**What are the advantages of ReLU over softmax in deep neural network?**

ReLU have the following properties:

* It has constant derivative. That make is very good for deep neural network which suffers from vanishing and explosion gradient when the gradient of the activation function is not constant because the gradient of a layer gets multiplied by the gradient of the next layer when you back-propagate it so that make it gets larger exponentially thus we call it exploding gradient if the gradient is bigger than one (thus not constant) or it will gets smaller exponentially thus we call it vanishing when the gradient is less than one.
* It is not differentiable on zero which makes not the very good choice when you want to output a continuas functions. But it is much more powerful when you what just to extract features of pattern recognition like in ConvNets for example.
* It has less capacity than the sigmoid function. The ReLu function is just a approximation function for **softplus**function which is the integral of the sigmoid function so that the first point is that it is approximation function which have less capacity of the original one:
* The other benefit of ReLUs is sparsity. Sparsity arises when a≤0. The more such units that exist in a layer the more sparse the resulting representation. Sigmoids on the other hand are always likely to generate some non-zero value resulting in dense representations. Sparse representations seem to be more beneficial than dense representations.
* An advantage to ReLU other than avoiding vanishing gradients problem is that it has much lower run time.





**What is difference between difference between Softmax function and Sigmoid function?**

Sigmoid Function Usage

* The Sigmoid function used for **binary classification** in logistic regression model.
* While creating artificial neurons sigmoid function used as the **activation function**.
* In statistics, the **sigmoid function graphs** are common as a cumulative distribution function.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | # Required Python Package  import numpy as np    def sigmoid(inputs):      """      Calculate the sigmoid for the give inputs (array)      :param inputs:      :return:      """      sigmoid\_scores = [1 / float(1 + np.exp(- x)) for x in inputs]      return sigmoid\_scores      sigmoid\_inputs = [2, 3, 5, 6]  print "Sigmoid Function Output :: {}".format(sigmoid(sigmoid\_inputs)) |

Softmax Function Usage

* Used in multiple classification logistic regression model.
* In building neural networks softmax functions used in different layer levels

# Required Python Package

import numpy as np

def softmax(inputs):

"""

Calculate the softmax for the give inputs (array)

:param inputs:

:return:

"""

return np.exp(inputs) / float(sum(np.exp(inputs)))

softmax\_inputs = [2, 3, 5, 6]

print "Softmax Function Output :: {}".format(softmax(softmax\_inputs))