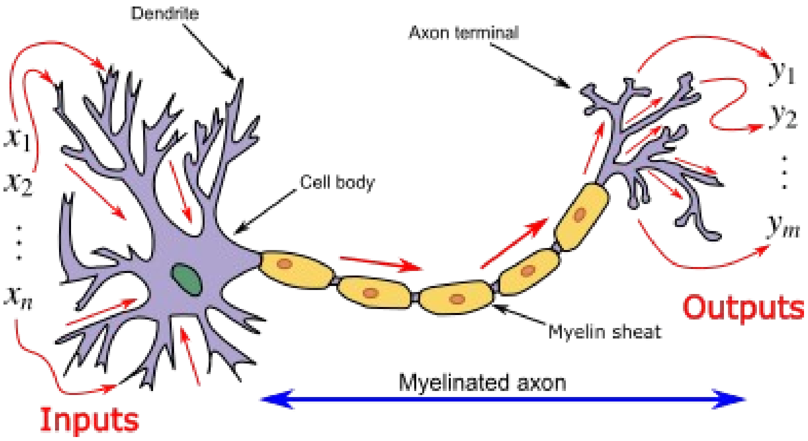
Neural Networks

Neural Nets are the foundation for the currently very strong and growing subfield of Machine Learning and AI called Deep Learning. In this unit we'll cover the basic principles that support the structure and operation of neural networks, and we will cover the fundamental training process called ***backpropagation*** that allows a network to learn a task from training data.

The neural networks covered here are fairly simple, but are easily generalized to the much more sophisticated and much larger Deep Learning networks used to carry out all kinds of exciting tasks, such as image classification, speech recognition, automated translation, and much more.

Neural networks are based on the view that the ability of the human brain to learn and carry out complex tasks depends on the organization and connection of billions of simple processing units (neurons). The idea being that the processing units themselves don't perform very complex tasks, but in their connections and interactions, complex behaviour can arise.

As a reminder, a biological neuron is a cell that consists of the following parts:



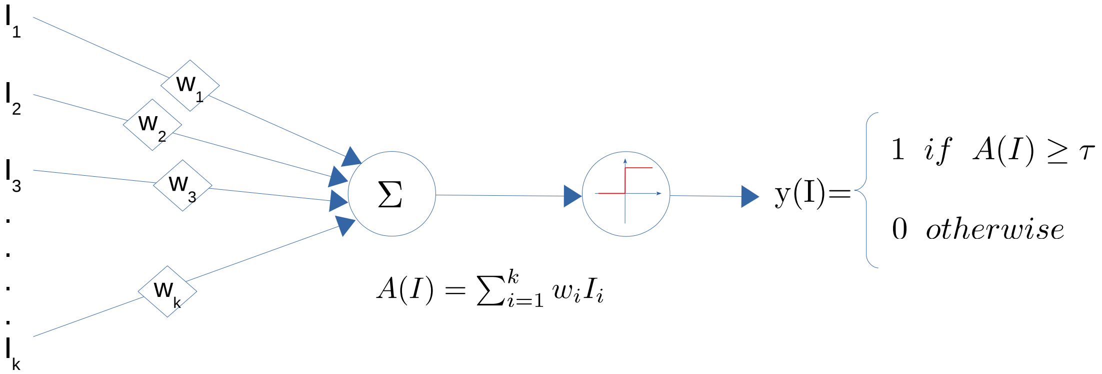
(Image credit: Wikipedia, by Egm4313.s12, CC SA 4.0)

A very simplified view of how a neuron operates is as follows:

- Inputs (the dendrites around the cell body) connect to other neurons
- At any give time, if the sum of the impulses arriving at the inputs to the neuron exceeds a threshold, the neuron *fires* (it sends a pulse down its axon, toward the output terminals connected to other neurons). Inputs can act as stimulatory or inhibitory signals, so their influence is either positive or negative

Of course, this is a very simplified view, in reality, the way a neuron responds is time dependent, and not entirely well understood.

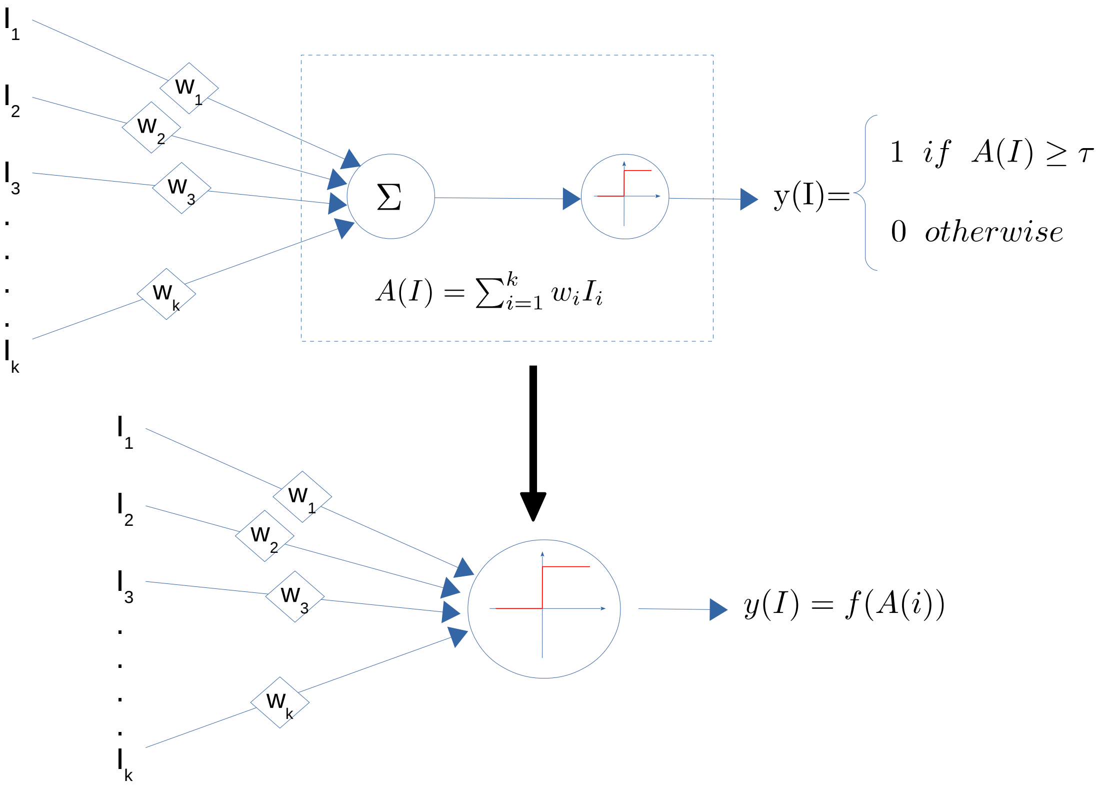
However, it is a good place to start if we're interested in building artificial neurons. The simplest (and still very much in use) model for how an ***artificial neuron*** may be implemented is the ***McCulloch-Pitts Neuron Model:***


I_1
I_2
I_3
I_k
W_1
W_2
W_3
W_k
Σ
$A(I) = \sum_{i=1}^{k} w_i I_i$
y(I)= $1 \; if \;\; A(I) \geq \tau$
$0 \;\; otherwise$

The McCulloch-Pitts model sees the artificial neuron as a simple unit that:

a) Computes a weighted sum of its input values (this becomes the ***activation*** A(I)).
b) Processes the activation through an ***activation unit*** which in the case above is a simple threshold function.
c) Determines the value of the output for the unit from the result of applying the activation function to the current value of A(I).

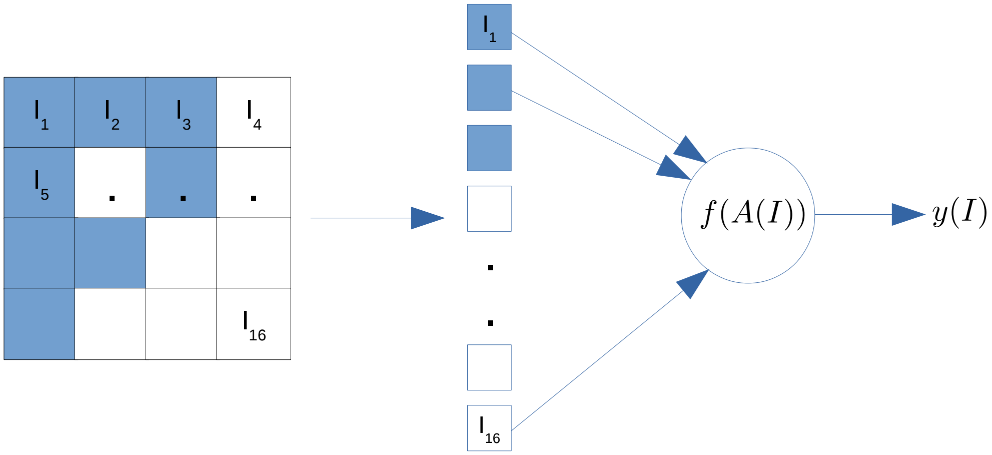
The McCulloch-Pitts model can be turned into a single unit in a neural network by grouping together the sum operation and the activation function:



The output of the neuron is the result of applying an ***activation function f(x) to the activation A(I)***. There are many different types of activation functions, but the operation of the neuron is the same.

A question that immediately comes to mind is ***what can we do with one of these artificial neurons?*** Let's see a simple example of what a single neuron could be ***trained*** to do.

Task: Given a small picture (4x4 pixels) of a text character, determine if the image corresponds to the letter 'P'.



Each pixel becomes one input to the neuron. Each pixel is connected with its own weight to the neuron, and the activation function in this case is a simple threshold.

We want the neuron to *fire* (output a 1) only if the input pattern corresponds to a 'P' such as the one shown above. How should we set the weights for the pixels?

Our first attempt may be to set the weights to 1.0 for all the weights connected to pixels that should be 'on' (blue in the image above) for the 'P' pattern, and we could make the weights 0.0 for all the remaining pixels. In the image above, the resulting A(I) would be equal to 8. We set the threshold of the activation function to 8, so the neuron will only fire when ***all*** of the pixels in the 'P' are set. If even a single pixel is 'off' that should be 'on', the neuron won't fire.

Is this good enough? What happens if a pixel that should be 'off' is 'on'? With the weights set to 1.0 or 0.0, the neuron won't care about the values of the 'off' pixels. It will recognize a fully blue image as a 'P'! this is not too great.

What if we set the weight to -1.0 for pixels that should be 'off'?

Now, if any pixel is 'off' that should be 'on', or if any pixel is 'on' that should be 'off', the value of A(I) will be below threshold and the neuron won't fire!

With a simple set of +1 and -1 weights, we can get this very simple neuron to recognize an input pattern perfectly. A larger image is simply a matter of adding more inputs to correspond to the number of input pixels. The neuron is equally capable of handling an input image of arbitrary size.

The example above shows that a very simple neuron can carry out a pattern recognition task. However, it should be clear to us that:

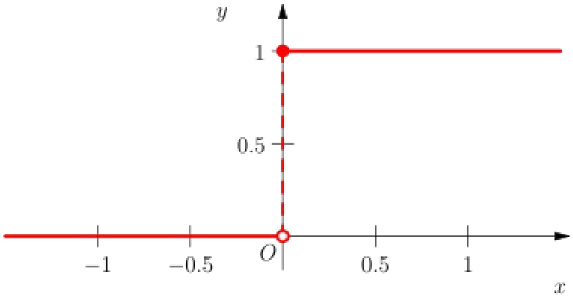
- On a realistic example, the pixels that are 'on' or 'off' for a given pattern won't be so perfectly defined. E.g. in hand-written optical character recognition we have to deal with variation between characters written by different people, as well as the differences in the individual characters written by the same user.
- The neuron has to ***learn*** which pixels should be more often 'on' or 'off', and which don't matter, and we want to be able to ***adjust*** the weights from a set of examples so the neuron can do the best possible job for any given pattern.

In what follows, we will look at the types of activation functions we can use to build artificial neurons, see how to organize a very simple network composed of multiple artificial neuron units, and learn how to train such a network to carry out a pattern recognition task.

Activation Functions

Several activation functions have been proposed and used in implementing neural networks

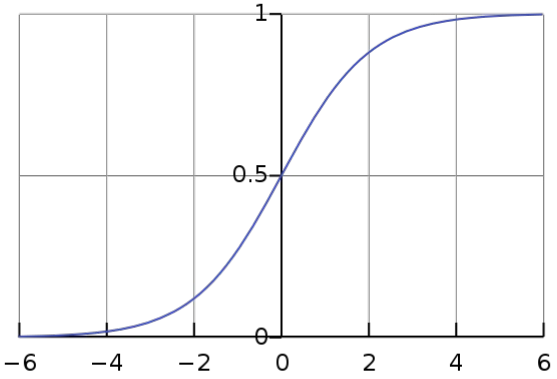
a) The ***threshold*** activation function:



(Image: Wikimedia Commons, by Lennart Kudling, public domain)

This function is 0 if the input is less than a threshold, and 1 otherwise. It is useful in understanding the idea of how an artificial neuron works, but not often used in practice – as it turns out, to implement the learning process we need a differentiable (well, mostly anyway) function.

b) The ***logistic*** function:

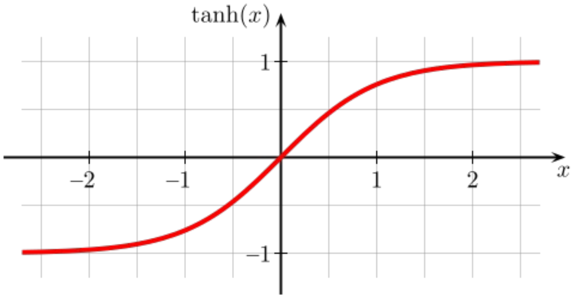


(Image: Wikimedia Commons, by Qef, public domain)

This function behaves like a soft threshold. For values much less than the threshold value, it outputs zero, for values significantly greater than the threshold, it outputs 1.0, and close to the threshold the output increases from 0.0 to 1.0.

The logistic function is given by: $f(x) = \frac{1}{1+e^{-x}}$

c) The hyperbolic tangent function:

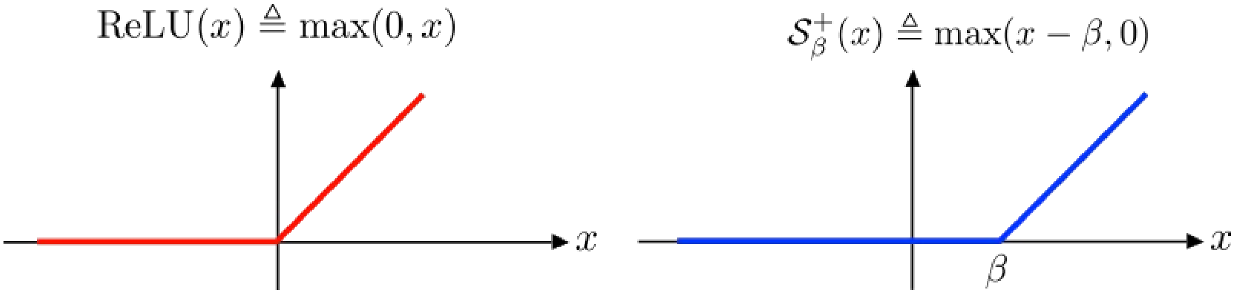


(Image: Wikimedia Commons, by Geek3, CC-SA 3.0)

This function has the same general shape as the *logistic* function, however, its values range from -1.0 to 1.0, which has advantages for certain tasks. Because of their similar 'S' shape, both the *logistic function* and the *hyperbolic tangent* are known as ***sigmoid functions***.

The hyperbolic tangent is given by: $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

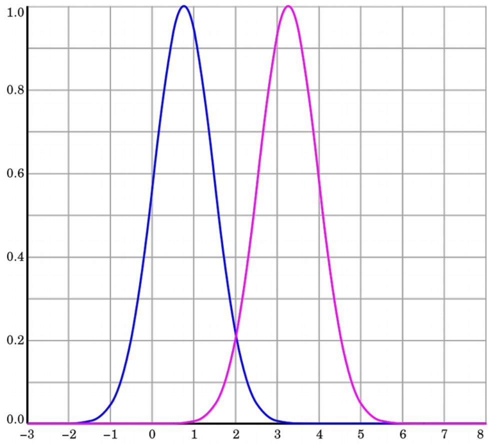
d) The Rectified Linear Unit (ReLU)



(Image: Wikimedia Commons, by Renanar2, CC-SA 4.0)

The rectified linear unit is also known as the *soft threshold* unit, and consists of two segments. Its value is 0 if the input is less than the threshold, and increases linearly with x if x is greater than the threshold. It is an often used function in larger neural networks such as those employed by Deep Learning.

e) Radial Basis Functions (RBFs)



(Image: Wikimedia Commons, by Pac72, public domain)

RBFs are Gaussian-shaped functions used for problems in which the spatial layout or pattern of an input is important. The width of the RBF is configurable, and can be fixed before the network is trained, or even tuned as part of the learning process.

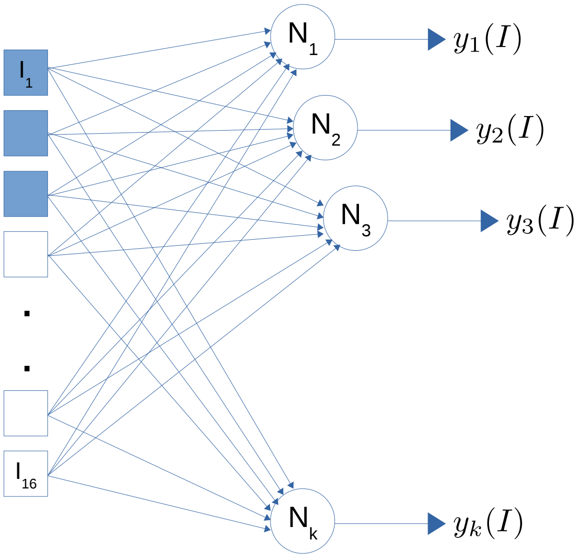
The above are common choices of activation functions for neural networks. Other choices exist, but whatever the form of the activation function, the fundamental architecture of the network and the training process remain the same. Let’s see how to build and train a simple neural network!

The Single Layer, Feed Forward Neural Network

A single-layer, feed forward network consists of a set of artificial neurons all connected to the same set of inputs, and all of which have their own output. A sample application for such a network is a system for recognizing hand-written characters. Each neuron is responsible for one of the characters we want to recognize, its output should be ***large*** when the corresponding character is present in the input, and ***small*** when a different character is shown.

By looking at the outputs of all the neurons, and picking the neuron with the largest output for a given input, we can identify the input character with high accuracy.

The structure of a single-layer feed-forward network is shown below:



Remember: Each connection from an input to a neuron has its own ***weight*** so in the network above we have a total of ***16*k*** weights that need to be adjusted to have the network perform the task we want. Notice that each neuron is actually independent from each other, there are no connections between neurons, so what we really have here is a set of independent neurons each of which will be trained to look for a different pattern. They only form a network in the sense that at the very end, we look at the output of all neurons to pick the one with the largest output as the 'correct' answer to which pattern is in the input.

The training process for Neural Nets is a type of reinforcement learning. The network's weights are adjusted based on the difference between what the network should have output given the input, and what the network actually output for that sample input. This requires us to have a possibly large ***training set*** comprised of many sample inputs, with the corresponding expected output value for the task we're training the network to solve. Such an ***annotated training set*** is a pre-requisite for training a neural network.

Given an input training dataset, the process for training the network is described by the following short algorithm:

```
Training Loop

Until  error on training set[1] is small

    For each input $I_i$ on training[2] set

        a) Feed-forward pass: Show the input $I_i$ to the network, and compute outputs for
           all neurons (on a multi-layer network, this is done layer by layer from those
           closest to the input and proceeding toward the output layer).
        b) At output layer - compute the error[3] between a neuron's output and the
           expected output for that neuron.
        c) Adjust the network weights so as to reduce error[4] on this training sample.
```

[1] There are several different measures of error commonly used in training neural networks. Here we will use ***squared error:*** $err_j(I_i) = (T_{j,i} - O_{j,i})^2$ - this is the error for ***output neuron j*** on ***input I_i*** , and it corresponds to the difference between the ***target output for this neuron on this input,*** and its ***actual output, squared.***

The error on the ***training set*** is defined simply as the sum of the error over all training samples:

$$TotalSquaredError = \sum_{i,j} err_j I_i$$

There are many other ways to quantify error on the training set, we could use RMS, or check total squared error for each output neuron separately. There are also many variations in functions used to evaluate error for a neuron’s output. For now, keep in mind that the point of this is: we measure (in some way) the difference between what the neuron should have output, and what it actually output, and we use that to determine how well the network is doing by accumulating or averaging error across all output neurons on a set of training samples.

So, when do we stop?

The total error on the training set may never decrease below an arbitrary threshold (the task may be too hard to solve well by the current network, or the input dataset may not be large enough, or a combination of both). So instead of setting a threshold on the total squared error, we can:

a) Run the training loop for a pre-specified number of iterations
b) Stop if after K iterations, the total squared error has only decreased slightly (the network is not improving a lot anymore)
c) Stop if a separate ***cross-validation*** step shows our network is not getting better on new data it hasn’t seen during training.

[2] The simplest training procedure loops over every input in the training set each time, and adjusts the weights after each input is processed. This works, but has a tendency to converge slowly to the network weights that yield good performance.

This happens because different training samples may adjust the same weight in opposite directions – training sample A says we should increase that weight’s value, training sample B just after says we should decrease it.

Over time, the network should converge to reasonable weights, but it may take a while since the weight update directions oscillate frequently.

One way to improve on this is to ***batch updates,*** that is, within the training loop, we group inputs into subsets of ***k,*** accumulate the squared error for each output neuron over the entire subset of ***k*** inputs, and then use this accumulated error to adjust the weights.

Because we’re now computing error over a subset of inputs, not just one, it is expected that the accumulated error will provide a more reliable estimate of the direction in which weights have to be adjusted in order to improve network performance.

Batch updating is almost always used in training neural nets. There’s one more improvement to be made to this process: ***Each subset of k inputs is chosen randomly, from the training set, each time.*** This ***stochastic batch update*** has the additional advantage that the network will be resilient to accidental patterns present in the input training data that occur when inputs are shown in a certain order. In machine learning, this is called ***over-fitting*** and is a general problem in which any learning algorithm starts to learn peculiarities of the training set that do not exist in other data, and because of it, don't work so well when new data is shown.

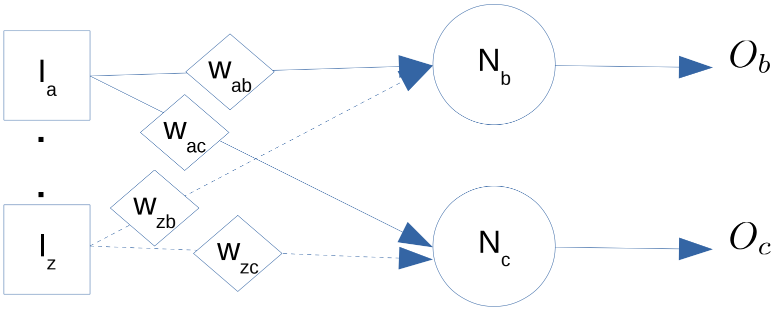
So, we can train the network one sample at a time, or we can use a more advanced training process such as ***stochastic batch updates***, which will likely help the network converge faster, and converge toward a configuration that is less likely to overfit.

[3] This is just the squared error for each output neuron, as described in [1], or whatever measure of difference is being used to train the network.

[4] The process of adjusting the weights is called ***back propagation***, and we will look at it in detail now.

Adjusting the network's weights: Error backpropagation

Let's consider a single-layer network, and let us look at how we would update ***one weight*** from an input to one of the neuron's in the network. The situation is shown in the image below:



This very simple network has two neurons, two outputs, and a multiple inputs. However, the same procedure below applies to single-layer networks with any number of inputs, and any number of outputs.

Adjusting a single weight

Let's look at weight w_{ab} in the network above. This links input a to neuron b. The idea is to look at the error in the output and determine in which direction we need to change this weight so as to make the output of the neuron closer to the expected output for this particular input.

As it turns out, the quantity we really want to know is: $\frac{\partial Err_b}{\partial w_{ab}}$, in other words, we need to know how the error for this neuron changes if we change the value of the weight w_{ab}.

We can apply the chain rule to break this down into manageable chunks:

$$\frac{\partial Err_b}{\partial w_{ab}} = \frac{\partial A(I)}{\partial w_{ab}} \cdot \frac{\partial O_b}{\partial A(I)} \cdot \frac{\partial Err_b}{\partial O_b}$$

Remember that A(I) is the ***activation*** for the neuron:

$$A(I) = \sum_i w_i \cdot I_i$$

Since we are using ***squared error,*** we have:

$$\frac{\partial Err_b}{\partial O_b} = -2(T_b - O_b)$$

Notice that we dropped a sub-index for input j, it is assumed we're computing this quantity for whatever current input is being shown to the network. Secondly, we will drop the factor of -2, since the update equation has its own scalar ***learning rate*** as we will see above, and we can account for any constants and sign changes with this single parameter.

Next, we have the term $\frac{\partial O_b}{\partial A(I)}$ which tells us ***how the output of the neuron changes*** when the ***activation*** for the neuron changes. If you think about it for a moment, you'll realize we're just asking for the ***partial derivative of the neuron's activation function*** with respect to its input. What this term is, then, depends on what type of activation function we are using, but common examples are:

Logistic activation: $\frac{\partial O_b}{\partial A(I)} = f(A(I)) \cdot (1 - f(A(I))$, where f(x) is the logistic function on x

Hyperbolic tangent activation: $\frac{\partial O_b}{\partial A(I)} = 1 - tanh^2(A(I))$

Finally, we have the term

$$\frac{\partial A(I)}{\partial w_{ab}} = \frac{\partial \sum_i w_{ib} I_i}{\partial w_{ab}} = I_a$$

That's because the only term that involves w_{ab} is the one for I_a.

Putting everything together, for output neurons using ***the logistic activation function*** we have:

$$\frac{\partial Err_b}{\partial w_{ab}} = I_a \cdot f(A(I)) \cdot (1 - f(A(I))) \cdot (T_b - O_b),$$

(remember we dropped a factor of -2) and the corresponding weight update is given by

$$w_{ab} = w_{ab} + \alpha \frac{\partial Err_b}{\partial w_{ab}}$$

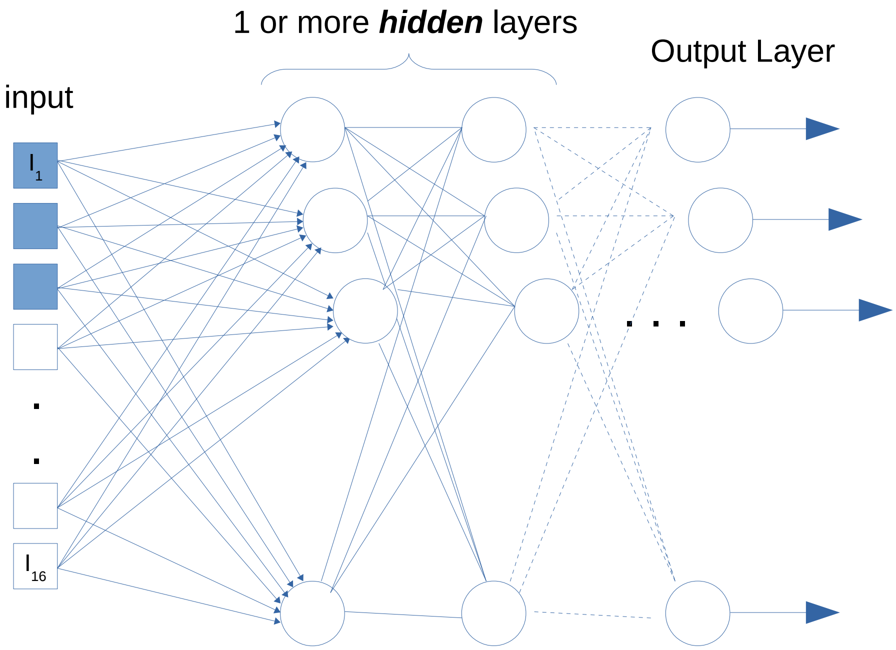
Here, the parameter α is the ***learning rate,*** and should be a small value for the network to converge.

For each input sample, or for each batch update we have to update ***every single weight in the network*** using the above update rule. You have seen this equation before in a slightly different form, it's just a form of ***gradient descent!*** We used it in Q-Learning to update the Q table and the policy. Here it will help us obtain the optimal network weights.

Note: If we are using ***batch updates***, then we are using ***batch gradient descent***, and if in addition we are using ***random subsets*** of inputs during each training round, we're then using ***stochastic gradient descent (SGD)***. Stochastic gradient descent is a very common optimization method for parameters in machine learning and AI.

What about multi-layer networks?

Single-layer networks can perform many tasks fairly well – you will train one for your assignment and see just how well it does in a character recognition task. However, for more complex problems we need ***crunchier networks***. A general multi-layer network has the following structure:



Now we have introduced one or more ***'hidden layers'***, they are called hidden because a network's user sees a black-box with only a set of outputs, and doesn't have access to the network structure so they can't tell what's actually in the box!

Each layer is ***fully-connected*** to the next one, that is, the outputs of the neurons in layer 1 act as input for the neurons in layer 2, and so on. It's worth thinking a moment about what neurons in hidden layers are doing:

Layer 1 neurons process input data and respond to ***patterns of interest*** in the input

Layer 2 neurons have access to ***interesting patterns*** also called ***features*** that were output by Layer 1 neurons, these ***are likely more informative*** than the actual inputs!
Layer 2 neurons learn ***patterns of interest in the patterns of interest*** output by Layer 1!

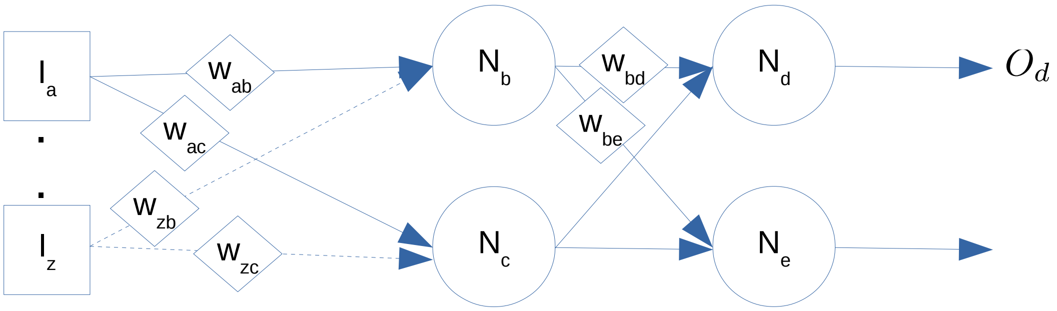
Layer 3 has access to ***even more informative features*** produced by Layer 2, and so on.

The net effect is each succeeding layer has access to more complex, more informative, and likely more useful features. The output layer is expected to be able to do a much better job since its input now consists of more powerful features abstracted by previous levels of the network.

You'll train a multi-layer network in your assignment, and compare its performance to the single layer network. It's important to remark at this point that ***deep learning*** refers to multi-layer networks with (some say 4, some say 6) or more layers. These very large networks require more work in terms of the numerical algorithms used to compute and apply weight updates, and allow for more complicated network topologies, but the same principles apply to deep learning nets that apply to 2 layer networks!

Updating weights in a hidden layer

We know how to update weights for a single layer network, it turns out that updating weights in a multi-layer network is not much more complicated. Let's look at an example:



We just added a layer to the network we used to study the weight updates for single-layer networks. The hidden layer has 2 neurons B and C, and the output layer now has two neurons D and E.

The weight updates for weights from the hidden layer to the output layer are identical to the single-layer weight updates above, except instead of using the input values, we use the output of the hidden network neurons.

The weight updates for weights from the ***input layer*** to the ***output layer*** are a bit more interesting. We have the same ***chain rule*** as before:

$$\frac{\partial Err_b}{\partial w_{ab}} = \frac{\partial A(I)}{\partial w_{ab}} \frac{\partial O_b}{\partial A(I)} \frac{\partial Err_b}{\partial O_b}$$

Notice it is identical to the equation we have before! Except, it's not clear how to obtain the last term here $\frac{\partial Err_b}{O_b}$ since neuron B is now ***not an output neuron*** so we ***don't have a target value*** for what its output should be. Furthermore, the output of neuron B now contributes to the inputs to neurons D and E, which means it somehow contributes to error values for these two neurons.

This means that a change in w_{ab} now will cause a change in the error for output neurons D and E!

Happily, we can take the chain rule a bit further, and if we do that, we'll find that:

$$\frac{\partial Err_b}{\partial O_b} = \sum_{j \in \ neurons \ connected \ to \ B} w_{bj} \frac{\partial f(x)}{\partial A_j(I)} \cdot (T_j - O_j)$$

Let's see what this sum is doing:

- The last two terms are the same as we had before for the 1-layer update: The derivative of the activation function for neuron *j*, and the difference between the target output for neuron *j* , and the actual output for neuron *j*.

- These are multiplied by the weight joining neuron *B* to neuron *j*.

- And we add up the error contributions from all neurons connected to B which will be affected by an update to the weight w_{ab}.

Can you see in the formula above how we have exactly the same update as for the 1-layer network?

The update for the weight itself is just like before:

$$w_{ab} = w_{ab} + \alpha \frac{\partial Err_b}{w_{ab}}$$

And we compute this for every weight feeding the hidden layer. The same update formula applies to weights in between any pair of hidden layers, or between the input and the first hidden layer. The only thing that changes is ***what is taken as the input for a neuron***, and ***which connected neurons contribute to the error computation.***

This completes the process of training a neural net with one or multiple layers. However there's a couple of technical issues that we need to think about:

Technical considerations in training a neural network

a) Initializing the weights

The initial values for the weights in the network are usually small random numbers (both positive and negative).

Keep in mind that this means there will be differences in the results obtained over different training runs! But these variations should be fairly small.

b) Weights becoming too large!

This is a problem known as exploding gradient, and is typical of networks trained with vanilla SGD or regular gradient descent. Some weights tend to grow in magnitude beyond anything reasonable and once they are huge the network is not working very well.

Keep an eye out for such exploding weights, and nuke them (reset to a small random value) if found. You may also think of making the learning rate smaller.

c) Vanishing gradient

The training procedure depends on having a meaningful gradient for all the partial derivatives involved in the weight updates. The part that comes from the derivative of the activation function can be particularly problematic.

For the ***sigmoid*** functions, tanh() and logistic, once we are at the part of the curve that is flat (either in the negative infinity direction, or the positive infinity direction), the curve is flat so the gradient is close to zero. What is happening here is the neurons have become ***saturated,*** their output value is pretty much the same if we change the input by even a significant amount.

The result is that for such neurons, any weight updates will be close to zero and the learning process becomes stuck.

For learning to happen, the neurons should be operating within the part of their response curve, away from saturation. We must keep an eye on whether the neurons are becoming saturated and if so we may want to change the learning rate (make it smaller), and also check for weights that have become too large.

We should also be careful what target value we want the neurons to achieve. If we ask neurons with a ***sigmoidal*** activation function to output either 1.0 or 0.0 as a target, we are directly encouraging the neurons to move toward the saturation region of their response curve. Better target values may be .2 and .8, for the logistic function, and -.8 and +.8 for the tanh().

Toward Deep Learning

The principles above apply to networks of any size. However, to build much larger networks with multiple layers, standard SGD and squared error become insufficient (this is due in part to problems like the vanishing gradient). The move toward Deep Learning required advances in multiple aspects of the network training process, including:

- Better methods to keep track of the gradient and to update network weights
- Different activation functions (the ReLU activation function does not saturate!)
- Different error (now usually called ***loss***) functions
- Much larger input datasets (Deep Learning goes hand in hand with Big Data)
- More computing power! (this is an expensive training process, both in terms of computation and memory requirements)
- A framework for setting up and training the network (and now we have several, such as Tensor Flow, Caffe, and so on).

However, the principles you learned here form the basis of all of those improvements, and understanding the fundamental structure and training process of the 1 and 2-layer networks we discussed in these notes will allow you to grasp how much more complex Deep Learning networks are organized and trained.

Now, let’s go have fun with the assignment training some neural nets!