There's one additional rule of thumb that helps for supervised learning problems. You can usually prevent over-fitting if you keep your number of neurons below:

Nh=Ns(α∗(Ni+No))Nh=Ns(α∗(Ni+No))

* NiNi = number of input neurons.
* NoNo = number of output neurons.
* NsNs = number of samples in training data set.
* αα = an arbitrary scaling factor usually 2-10.

[Others recommend](http://www.solver.com/training-artificial-neural-network-intro) setting alphaalpha to a value between 5 and 10, but I find a value of 2 will often work without overfitting. You can think of alpha as the effective branching factor or number of nonzero weights for each neuron. Dropout layers will bring the "effective" branching factor way down from the actual mean branching factor for your network.

As explained by this [excellent NN Design text](http://hagan.okstate.edu/NNDesign.pdf#page=469), you want to limit the number of free parameters in your model (i.e. its [degree](https://stats.stackexchange.com/q/57027/15974) or the number of nonzero weights) to a small portion of the degrees of freedom in your data. The degrees of freedom in your data is the number samples \* degrees of freedom (dimensions) in each sample or Ns∗(Ni+No)Ns∗(Ni+No) (assuming they're all independent). So αα is a way to indicate how general you want your model to be, or how much you want to prevent overfitting.

For an automated procedure you'd start with an alpha of 2 (twice as many degrees of freedom in your training data as your model) and work your way up to 10 if the error (loss) for your training dataset is significantly smaller than for your test dataset.

**The Hidden Layers**

So those few rules set the number of layers and size (neurons/layer) for both the input and output layers. That leaves the hidden layers.

How many hidden layers? Well if your data is linearly separable (which you often know by the time you begin coding a NN) then you don't need any hidden layers at all. Of course, you don't need an NN to resolve your data either, but it will still do the job.

Beyond that, as you probably know, there's a mountain of commentary on the question of hidden layer configuration in NNs (see the insanely thorough and insightful [NN FAQ](http://www.faqs.org/faqs/ai-faq/neural-nets/part1/preamble.html) for an [excellent summary](http://www.faqs.org/faqs/ai-faq/neural-nets/part1/preamble.html) of that commentary). One issue within this subject on which there is a consensus is the performance difference from adding additional hidden layers: the situations in which performance improves with a second (or third, etc.) hidden layer are very few. *One hidden layer is sufficient for the large majority of problems.*

So what about size of the hidden layer(s)--how many neurons? There are some empirically-derived rules-of-thumb, of these, the most commonly relied on is '*the optimal size of the hidden layer is usually between the size of the input and size of the output layers*'. Jeff Heaton, author of [Introduction to Neural Networks in Java](https://www.heatonresearch.com/book/) offers a few more.

In sum, for most problems, one could probably get decent performance (even without a second optimization step) by setting the hidden layer configuration using just two rules: (i) number of hidden layers equals one; and (ii) the number of neurons in that layer is the mean of the neurons in the input and output layers.

There are many rule-of-thumb methods for determining the correct number of neurons to use in the hidden layers, such as the following:

* The number of hidden neurons should be between the size of the input layer and the size of the output layer.
* The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
* The number of hidden neurons should be less than twice the size of the input layer.

These three rules provide a starting point for you to consider. Ultimately, the selection of an architecture for your neural network will come down to trial and error. But what exactly is meant by trial and error? You do not want to start throwing random numbers of layers and neurons at your network. To do so would be very time consuming. Chapter 8, “Pruning a Neural Network” will explore various ways to determine an optimal structure for a neural network.