Maxout. Other types of units have been proposed that do not have the functional form $f(w^T x + b)$ where a non-linearity is applied on the dot product between the weights and the data. One relatively popular choice is the Maxout neuron (introduced recently by Goodfellow et al.) that generalizes the ReLU and its leaky version. The Maxout neuron computes the function $\max(w_1^T x + b_1, w_2^T x + b_2)$. Notice that both ReLU and Leaky ReLU are a special case of this form (for example, for ReLU we have $w_1, b_1 = 0$). The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks (dying ReLU). However, unlike the ReLU neurons it doubles the number of parameters for every single neuron, leading to a high total number of parameters.

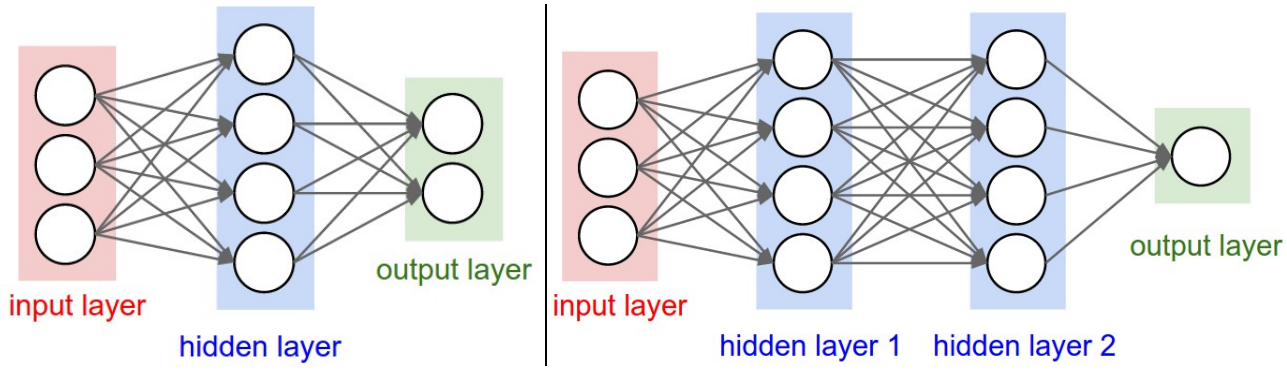
This concludes our discussion of the most common types of neurons and their activation functions. As a last comment, it is very rare to mix and match different types of neurons in the same network, even though there is no fundamental problem with doing so.

TLDR: "*What neuron type should I use?*" Use the ReLU non-linearity, be careful with your learning rates and possibly monitor the fraction of "dead" units in a network. If this concerns you, give Leaky ReLU or Maxout a try. Never use sigmoid. Try tanh, but expect it to work worse than ReLU/Maxout.

Neural Network architectures

Layer-wise organization

Neural Networks as neurons in graphs. Neural Networks are modeled as collections of neurons that are connected in an acyclic graph. In other words, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network. Instead of an amorphous blobs of connected neurons, Neural Network models are often organized into distinct layers of neurons. For regular neural networks, the most common layer type is the **fully-connected layer** in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections. Below are two example Neural Network topologies that use a stack of fully-connected layers:



Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs. **Right:** A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

Naming conventions. Notice that when we say N-layer neural network, we do not count the input layer. Therefore, a single-layer neural network describes a network with no hidden layers (input directly mapped to output). In that sense, you can sometimes hear people say that logistic regression or SVMs are simply a special case of single-layer Neural Networks. You may also hear these networks interchangeably referred to as *"Artificial Neural Networks"* (ANN) or *"Multi-Layer Perceptrons"* (MLP). Many people do not like the analogies between Neural Networks and real brains and prefer to refer to neurons as *units*.

Output layer. Unlike all layers in a Neural Network, the output layer neurons most commonly do not have an activation function (or you can think of them as having a linear identity activation function). This is because the last output layer is usually taken to represent the class scores (e.g. in classification), which are arbitrary real-valued numbers, or some kind of real-valued target (e.g. in regression).

Sizing neural networks. The two metrics that people commonly use to measure the size of neural networks are the number of neurons, or more commonly the number of parameters. Working with the two example networks in the above picture:

- The first network (left) has 4 + 2 = 6 neurons (not counting the inputs), [3 x 4] + [4 x 2] = 20 weights and 4 + 2 = 6 biases, for a total of 26 learnable parameters.
- The second network (right) has 4 + 4 + 1 = 9 neurons, [3 x 4] + [4 x 4] + [4 x 1] = 12 + 16 + 4 = 32 weights and 4 + 4 + 1 = 9 biases, for a total of 41 learnable parameters.

To give you some context, modern Convolutional Networks contain on orders of 100 million parameters and are usually made up of approximately 10-20 layers (hence *deep learning*). However, as we will see the number of *effective* connections is significantly greater due to parameter sharing. More on this in the Convolutional Neural Networks module.

Example feed-forward computation

Repeated matrix multiplications interwoven with activation function. One of the primary reasons that Neural Networks are organized into layers is that this structure makes it very simple and efficient to evaluate Neural Networks using matrix vector operations. Working with the example three-layer neural network in the diagram above, the input would be a [3x1] vector. All connection strengths for a layer can be stored in a single matrix. For example, the first hidden layer's weights `W1` would be of size [4x3], and the biases for all units would be in the vector `b1`, of size [4x1]. Here, every single neuron has its weights in a row of `W1`, so the matrix vector multiplication `np.dot(W1,x)` evaluates the activations of all neurons in that layer. Similarly, `W2` would be a [4x4] matrix that stores the connections of the second hidden layer, and `W3` a [1x4] matrix for the last (output) layer. The full forward pass of this 3-layer neural network is then simply three matrix multiplications, interwoven with the application of the activation function:

```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

In the above code, `W1,W2,W3,b1,b2,b3` are the learnable parameters of the network. Notice also that instead of having a single input column vector, the variable `x` could hold an entire batch of training data (where each input example would be a column of `x`) and then all examples would be efficiently evaluated in parallel. Notice that the final Neural Network layer usually doesn't have an activation function (e.g. it represents a (real-valued) class score in a classification setting).

> *The forward pass of a fully-connected layer corresponds to one matrix multiplication followed by a bias offset and an activation function.*

Representational power

One way to look at Neural Networks with fully-connected layers is that they define a family of functions that are parameterized by the weights of the network. A natural question that arises is: What is the representational power of this family of functions? In particular, are there functions that cannot be modeled with a Neural Network?

It turns out that Neural Networks with at least one hidden layer are *universal approximators*. That is, it can be shown (e.g. see *Approximation by Superpositions of Sigmoidal Function* from 1989

(pdf), or this intuitive explanation from Michael Nielsen) that given any continuous function $f(x)$ and some $\epsilon > 0$, there exists a Neural Network $g(x)$ with one hidden layer (with a reasonable choice of non-linearity, e.g. sigmoid) such that $\forall x, \mid f(x) - g(x) \mid < \epsilon$. In other words, the neural network can approximate any continuous function.

If one hidden layer suffices to approximate any function, why use more layers and go deeper? The answer is that the fact that a two-layer Neural Network is a universal approximator is, while mathematically cute, a relatively weak and useless statement in practice. In one dimension, the "sum of indicator bumps" function $g(x) = \sum_i c_i 1(a_i < x < b_i)$ where a, b, c are parameter vectors is also a universal approximator, but noone would suggest that we use this functional form in Machine Learning. Neural Networks work well in practice because they compactly express nice, smooth functions that fit well with the statistical properties of data we encounter in practice, and are also easy to learn using our optimization algorithms (e.g. gradient descent). Similarly, the fact that deeper networks (with multiple hidden layers) can work better than a single-hidden-layer networks is an empirical observation, despite the fact that their representational power is equal.

As an aside, in practice it is often the case that 3-layer neural networks will outperform 2-layer nets, but going even deeper (4,5,6-layer) rarely helps much more. This is in stark contrast to Convolutional Networks, where depth has been found to be an extremely important component for a good recognition system (e.g. on order of 10 learnable layers). One argument for this observation is that images contain hierarchical structure (e.g. faces are made up of eyes, which are made up of edges, etc.), so several layers of processing make intuitive sense for this data domain.

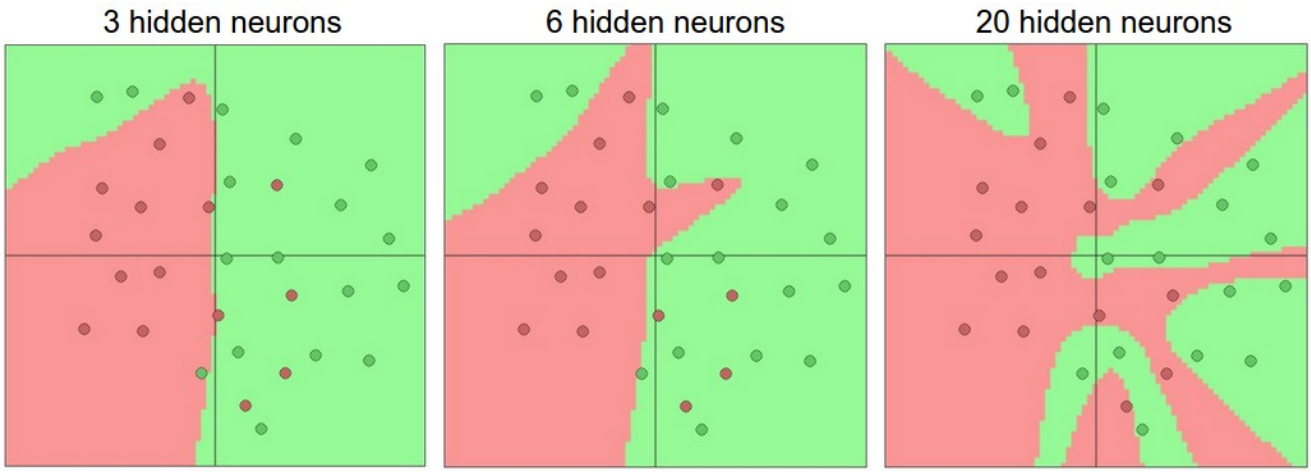
The full story is, of course, much more involved and a topic of much recent research. If you are interested in these topics we recommend for further reading:

- Deep Learning book in press by Bengio, Goodfellow, Courville, in particular Chapter 6.4.
- Do Deep Nets Really Need to be Deep?
- FitNets: Hints for Thin Deep Nets

Setting number of layers and their sizes

How do we decide on what architecture to use when faced with a practical problem? Should we use no hidden layers? One hidden layer? Two hidden layers? How large should each layer be? First, note that as we increase the size and number of layers in a Neural Network, the **capacity** of the network increases. That is, the space of representable functions grows since the neurons can collaborate to express many different functions. For example, suppose we had a binary

classification problem in two dimensions. We could train three separate neural networks, each with one hidden layer of some size and obtain the following classifiers:



Larger Neural Networks can represent more complicated functions. The data are shown as circles colored by their class, and the decision regions by a trained neural network are shown underneath. You can play with these examples in this ConvNetsJS demo.

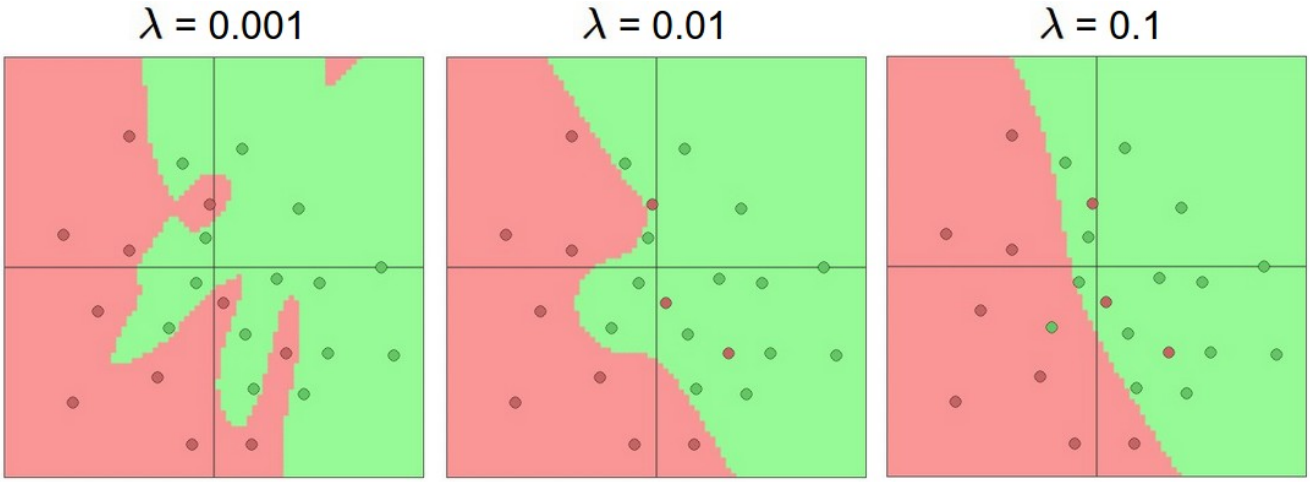
In the diagram above, we can see that Neural Networks with more neurons can express more complicated functions. However, this is both a blessing (since we can learn to classify more complicated data) and a curse (since it is easier to overfit the training data). **Overfitting** occurs when a model with high capacity fits the noise in the data instead of the (assumed) underlying relationship. For example, the model with 20 hidden neurons fits all the training data but at the cost of segmenting the space into many disjoint red and green decision regions. The model with 3 hidden neurons only has the representational power to classify the data in broad strokes. It models the data as two blobs and interprets the few red points inside the green cluster as **outliers** (noise). In practice, this could lead to better **generalization** on the test set.

Based on our discussion above, it seems that smaller neural networks can be preferred if the data is not complex enough to prevent overfitting. However, this is incorrect - there are many other preferred ways to prevent overfitting in Neural Networks that we will discuss later (such as L2 regularization, dropout, input noise). In practice, it is always better to use these methods to control overfitting instead of the number of neurons.

The subtle reason behind this is that smaller networks are harder to train with local methods such as Gradient Descent: It's clear that their loss functions have relatively few local minima, but it turns out that many of these minima are easier to converge to, and that they are bad (i.e. with high loss). Conversely, bigger neural networks contain significantly more local minima, but these minima turn out to be much better in terms of their actual loss. Since Neural Networks are non-convex, it is hard to study these properties mathematically, but some attempts to understand these objective functions have been made, e.g. in a recent paper The Loss Surfaces of Multilayer

Networks. In practice, what you find is that if you train a small network the final loss can display a good amount of variance - in some cases you get lucky and converge to a good place but in some cases you get trapped in one of the bad minima. On the other hand, if you train a large network you'll start to find many different solutions, but the variance in the final achieved loss will be much smaller. In other words, all solutions are about equally as good, and rely less on the luck of random initialization.

To reiterate, the regularization strength is the preferred way to control the overfitting of a neural network. We can look at the results achieved by three different settings:



The effects of regularization strength: Each neural network above has 20 hidden neurons, but changing the regularization strength makes its final decision regions smoother with a higher regularization. You can play with these examples in this ConvNetsJS demo.

The takeaway is that you should not be using smaller networks because you are afraid of overfitting. Instead, you should use as big of a neural network as your computational budget allows, and use other regularization techniques to control overfitting.

Summary

In summary,

- We introduced a very coarse model of a biological **neuron**
- We discussed several types of **activation functions** that are used in practice, with ReLU being the most common choice
- We introduced **Neural Networks** where neurons are connected with **Fully-Connected layers** where neurons in adjacent layers have full pair-wise connections, but neurons within a layer are not connected.

- We saw that this layered architecture enables very efficient evaluation of Neural Networks based on matrix multiplications interwoven with the application of the activation function.
- We saw that that Neural Networks are **universal function approximators**, but we also discussed the fact that this property has little to do with their ubiquitous use. They are used because they make certain "right" assumptions about the functional forms of functions that come up in practice.
- We discussed the fact that larger networks will always work better than smaller networks, but their higher model capacity must be appropriately addressed with stronger regularization (such as higher weight decay), or they might overfit. We will see more forms of regularization (especially dropout) in later sections.

Additional References

- deeplearning.net tutorial with Theano
- ConvNetJS demos for intuitions
- Michael Nielsen's tutorials

---

cs231n
cs231n
karpathy@cs.stanford.edu