**Neural Networks: Structure**

If you recall from the [Feature Crosses unit](https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture), the following classification problem is nonlinear:

**Chart, bubble chart

Description automatically generated**

"**Nonlinear**" means that you can't accurately predict a label with a model of the form b+w1x1+w2x2.

In other words, the "**decision surface**" is not a line. Previously, we looked at [feature crosses](https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture) as one possible approach to modeling nonlinear problems.

**Chart, scatter chart

Description automatically generated**

The data set shown in Figure 2 can't be solved with a linear model.

To see how neural networks might help with nonlinear problems, let's start by representing a linear model as a graph:

Diagram

Description automatically generated with medium confidence

Each blue circle represents an input feature, and the green circle represents the weighted sum of the inputs.

How can we alter this model to improve its ability to deal with nonlinear problems?

**Hidden Layers**

In the model represented by the following graph, we've added a "hidden layer" of intermediary values. Each yellow node in the hidden layer is a weighted sum of the blue input node values. The output is a weighted sum of the yellow nodes.

Diagram

Description automatically generated

Is this model linear? Yes—its output is still a linear combination of its inputs.

In the model represented by the following graph, we've added a second hidden layer of weighted sums.

Diagram

Description automatically generated

Is this model still linear? Yes, it is. When you express the output as a function of the input and simplify, you get just another weighted sum of the inputs. This sum won't effectively model the nonlinear problem in Figure 2.

**Activation Functions**

To model a nonlinear problem, we can directly **introduce a nonlinearity**. We can pipe each hidden layer node through a nonlinear function.

In the model represented by the following graph, the value of each node in Hidden Layer 1 is transformed by a nonlinear function before being passed on to the weighted sums of the next layer. This nonlinear function is called the **activation function**.

Diagram

Description automatically generated

Now that we've added an activation function, adding layers has more impact. Stacking nonlinearities on nonlinearities lets us model very complicated relationships between the inputs and the predicted outputs.

**In brief, each layer is effectively learning a more complex, higher-level function over the raw inputs.**

If you'd like to develop more intuition on how this works, see [Chris Olah's excellent blog post](http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/).

**Common Activation Functions**

1. **Sigmoid Activation Function**

The following **sigmoid activation function** converts the weighted sum to a value between 0 and 1.

Chart, line chart

Description automatically generated

1. **Rectified Linear Unit**

The following rectified linear unit activation function (or ReLU, for short) often works a little better than a smooth function like the sigmoid, while also being significantly easier to compute.



The superiority of ReLU is based on empirical findings, probably driven by ReLU having a more useful range of responsiveness. A sigmoid's responsiveness falls off relatively quickly on both sides.

Chart, line chart

Description automatically generated

In fact, any mathematical function can serve as an activation function. Suppose that σ represents our activation function (Relu, Sigmoid, or whatever). Consequently, the value of a node in the network is given by the following formula:

Diagram, schematic

Description automatically generated

TensorFlow provides out-of-the-box support for many activation functions. You can find these activation functions within TensorFlow's [list of wrappers for primitive neural network operations](https://www.tensorflow.org/api_docs/python/tf/nn). That said, we still recommend starting with ReLU.

**Summary**

Now our model has all the standard components of what people usually mean when they say "neural network":

* A set of nodes, analogous to neurons, organized in layers.
* A set of weights representing the connections between each neural network layer and the layer beneath it. The layer beneath may be another neural network layer, or some other kind of layer.
* A set of biases, one for each node.
* An activation function that transforms the output of each node in a layer. Different layers may have different activation functions.

A caveat: neural networks aren't necessarily always better than feature crosses, but neural networks do offer a flexible alternative that works well in many cases.

**Key Terms**

|  |  |
| --- | --- |
|  [activation function](https://developers.google.com/machine-learning/glossary#activation_function) |  [hidden layer](https://developers.google.com/machine-learning/glossary#hidden_layer) |
|  [neural network](https://developers.google.com/machine-learning/glossary#neural_network) |  [neuron](https://developers.google.com/machine-learning/glossary#neuron) |
|  [rectified linear unit (ReLU)](https://developers.google.com/machine-learning/glossary#ReLU) |  [sigmoid function](https://developers.google.com/machine-learning/glossary#sigmoid_function) |

**Training Neural Networks**

**Backpropagation** is the most common training algorithm for neural networks.

It makes gradient descent feasible for multi-layer neural networks.

**TensorFlow handles backpropagation automatically**, so you don't need a deep understanding of the algorithm.

To get a sense of how it works, walk through the following: [Backpropagation algorithm visual explanation](https://developers.google.com/machine-learning/crash-course/backprop-scroll).

As you scroll through the preceding explanation, note the following:

* How **data flows through the graph**.
* How dynamic programming lets us avoid computing exponentially many paths through the graph. Here "**dynamic programming**" just means recording intermediate results on the forward and backward passes.

**Training Neural Networks: Best Practices**

This section explains backpropagation's failure cases and the most common way to regularize a neural network.

**Failure Cases**

There are a number of common ways for backpropagation to go wrong.

1. **Vanishing Gradients**

* **The gradients for the lower layers (closer to the input) can become very small.**
* In deep networks, computing these gradients can involve taking the product of many small terms.
* When the gradients vanish toward 0 for the lower layers, these layers train very slowly, or not at all.
* The ReLU activation function can help prevent vanishing gradients.

1. **Exploding Gradients**

* If the weights in a network are very large, then the gradients for the lower layers involve products of many large terms.
* **In this case you can have exploding gradients: gradients that get too large to converge.**
* Batch normalization can help prevent exploding gradients, as can lowering the learning rate.

1. **Dead ReLU Units**

* **Once the weighted sum for a ReLU unit falls below 0, the ReLU unit can get stuck.**
* It outputs 0 activation, contributing nothing to the network's output, and gradients can no longer flow through it during backpropagation.
* With a source of gradients cut off, the input to the ReLU may not ever change enough to bring the weighted sum back above 0.
* Lowering the learning rate can help keep ReLU units from dying.

1. **Dropout Regularization**

- **Yet another form of regularization, called Dropout, is useful for neural networks**.

It works by randomly "dropping out" unit activations in a network for a single gradient step. The more you drop out, the stronger the regularization:

* 0.0 = No dropout regularization.
* 1.0 = Drop out everything. The model learns nothing.
* Values between 0.0 and 1.0 = More useful.

**Key Terms**

|  |  |
| --- | --- |
|  [activation function](https://developers.google.com/machine-learning/glossary#activation_function) |  [backpropagation](https://developers.google.com/machine-learning/glossary#backpropagation) |
|  [dropout regularization](https://developers.google.com/machine-learning/glossary#dropout_regularization) |  [gradient descent](https://developers.google.com/machine-learning/glossary#gradient_descent) |
|  [rectified linear unit (ReLU)](https://developers.google.com/machine-learning/glossary#ReLU) | |

**Multi-Class Neural Networks: Softmax**

Recall that logistic regression produces a decimal between 0 and 1.0.

For example, a logistic regression output of 0.8 from an email classifier suggests an 80% chance of an email being spam and a 20% chance of it being not spam. Clearly, the sum of the probabilities of an email being either spam or not spam is 1.0.

Softmax extends this idea into a multi-class world. That is, Softmax assigns decimal probabilities to each class in a multi-class problem. Those decimal probabilities must add up to 1.0. This additional constraint helps training converge more quickly than it otherwise would.

For example, returning to the image analysis we saw in Figure 1, Softmax might produce the following likelihoods of an image belonging to a particular class:

Softmax is implemented through a neural network layer just before the output layer. The Softmax layer must have the same number of nodes as the output layer.

Table

Description automatically generated

Diagram

Description automatically generated

Text, letter

Description automatically generated

**Softmax Options**

Consider the following variants of Softmax:

* **Full Softmax** is the Softmax we've been discussing; that is, Softmax calculates a probability for every possible class.
* **Candidate sampling** means that Softmax calculates a probability for all the positive labels but only for a random sample of negative labels.

For example, if we are interested in determining whether an input image is a beagle or a bloodhound, we don't have to provide probabilities for every non-doggy example.

Full Softmax is fairly cheap when the number of classes is small but becomes prohibitively expensive when the number of classes climbs.

Candidate sampling can improve efficiency in problems having a large number of classes.

**One Label vs. Many Labels**

Softmax assumes that each example is a member of exactly one class. Some examples, however, can simultaneously be a member of multiple classes. For such examples:

* You may not use Softmax.
* You must rely on multiple logistic regressions.

For example, suppose your examples are images containing exactly one item—a piece of fruit. Softmax can determine the likelihood of that one item being a pear, an orange, an apple, and so on. If your examples are images containing all sorts of things—bowls of different kinds of fruit—then you'll have to use multiple logistic regressions instead.

**Key Terms**

|  |  |
| --- | --- |
|  [candidate sampling](https://developers.google.com/machine-learning/glossary#candidate_sampling) |  [logistic regression](https://developers.google.com/machine-learning/glossary#logistic_regression) |
|  [multi-class](https://developers.google.com/machine-learning/glossary#multi-class) |  [softmax](https://developers.google.com/machine-learning/glossary#softmax) |