Article from

Predictive Analytics and Futurism

December 2015
Issue 12

The Third Generation of Neural Networks

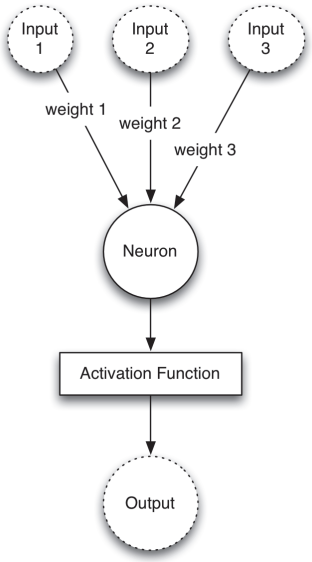
By Jeff Heaton

Neural networks are the phoenix of artificial intelligence. Right now neural networks are rising from the ashes for the third time since their introduction in the 1940s. There are many design decisions that a neural network practitioner must make. Because of their long history, there is a diverse amount of information about the architecture of neural networks. Because neural networks have essentially been invented (and reinvented) three times, much of this information is contradictory. This article presents what most current research dictates about how neural networks should be architected in 2015.

The goal of this article is to provide a tour of the most current technologies in the field of neural networks. A rigorous discussion of why these methods are effective is beyond both the scope, and space requirements, of this article. However, citations are provided to lead you to papers that provide justifications for the architectural decisions advocated by this article.

At the most abstract level, a neural network is still the weighted summation of its inputs, applied to an activation/transfer function, as shown in Figure 1.

FIGURE 1: SINGLE UNIT OF A NEURAL NETWORK



The above unit is still calculated using Equation 1, which has been the same formula since the first generation of neural networks.

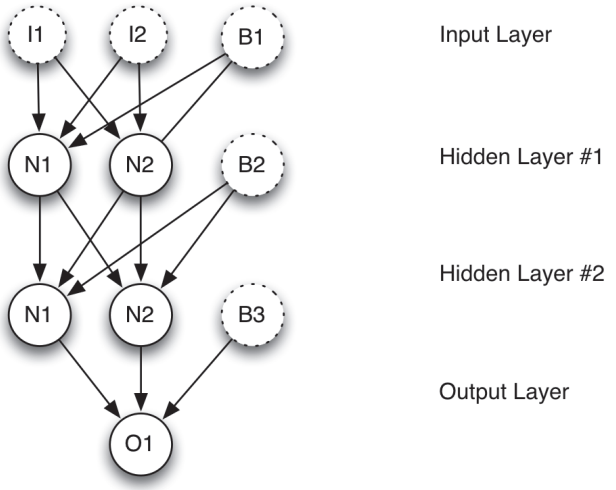
Equation 1: Neural Network Calculation

$$f(x_i, w_i) = \phi(\sum_i (w_i \cdot x_i))$$

The neural network output vector is dependent upon the input vector (x), the weights (w), and choice of activation function (phi, φ). Most implementations also use bias neurons that essentially become the y-intercept. To implement bias, most neural networks add a one to the x-vector and the bias-value to the weight vector. These values are both added at the beginning of these vectors. This is effectively the same as adding the bias/intercept term to the equation with a coefficient of one.

When these units are connected together, third generation neural networks still look the same as before. Figure 2 shows a two-input, single output neural network with two hidden layers.

FIGURE 2: MULTILAYER FEEDFORWARD NETWORK



The above diagram shows how the biases (indicated by B's) are added to each of the layers.

NUMBER OF LAYERS

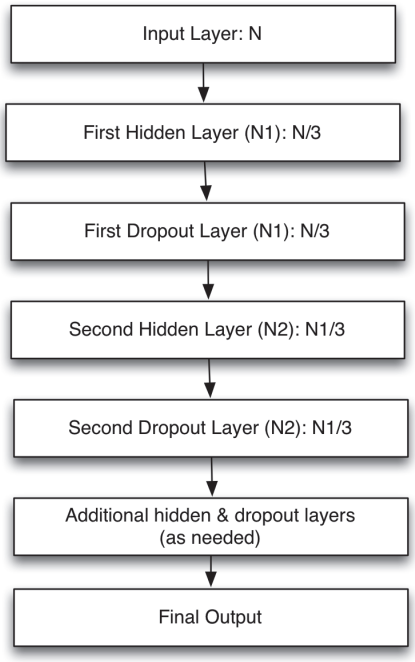
How many layers and how many hidden neurons has always been the primary question of the neural network practitioner. There is research that indicates that a single hidden layer in a neural network can approximate any function (Hornik, 1991). Because of this it is extremely tempting to use a single hidden layer neural

network for all problems. For several years, this was the suggested advice. However, just because a single layer network can, in theory, learn anything, the universal approximation theorem does not say anything about how easy it will be to learn. Additional hidden layers make problems easier to learn because they provide the hierarchical abstraction that is an inherent component in the human neocortex. Additional hidden layers are great, but the problem has been that we had no means of training such deep networks.

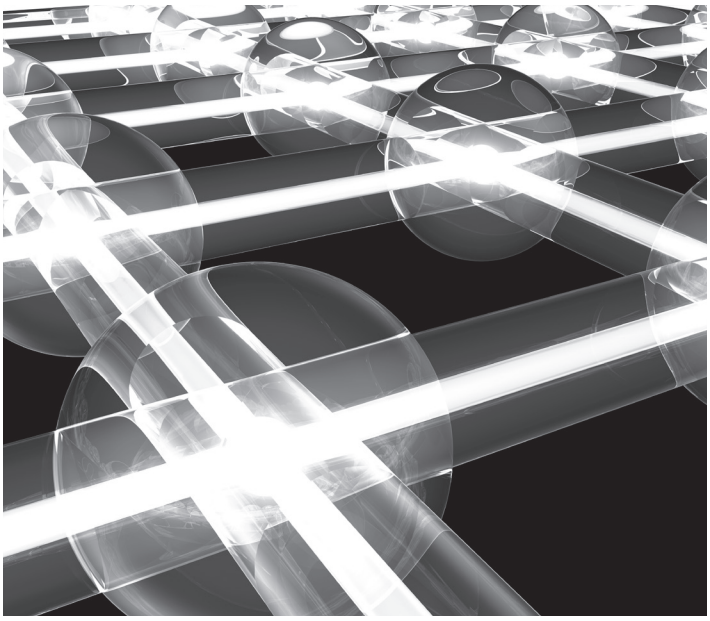
Deep learning is a very general term that describes a basket of technologies that allow neural networks, with more than two hidden layers, to be trained. Initially methods were discovered to train a deep belief neural network (DBNN), using clever techniques based on Gibbs sampling.[1] However, DBNN's can only accept binary inputs for classification. DBNN's showed the potential of deep learning and further research discovered the changes necessary to allow regular deep feedforward neural networks to be trained as well.

A deep modern neural network appears in Figure 3.

FIGURE 3: DEEP NEURAL NETWORK



The above diagram shows how additional pairs of hidden and dropout layers are added. These dropout layers, which help to avoid overfitting, will be discussed later in the article. Hidden layers and dropout layers usually occur in pairs. These hidden layers



are often called dense layers, because every neuron is connected to the next layer. Prior to the third generation of neural networks, every layer was dense. Dropout layers are not dense, as will be demonstrated later. You will also notice that the layers of the neural network decrease in their number of neurons. This forces the neural network to learn more and more abstract features of the input as the layers become deeper.

HIDDEN ACTIVATION FUNCTIONS

For years the choice of activation function for the hidden layers of a neural network was a choice between the two most common sigmoidal functions: the logistic and the hyperbolic tangent. Unfortunately, all sigmoidal (s-shaped) activation functions are difficult to train for deep neural networks. Because of this sigmoidal activation functions have largely fallen out of favor for neural networks. The activation function that has replaced them is the rectified linear unit (ReLU). The very simple equation for the ReLU is shown in Equation 2.

Equation 2: Rectified Linear Unit (ReLU)

$$\phi(x) = \max(0, x)$$

There are many papers written that provide more rigorous (Nair & Hinton, 2010) descriptions of the superiority of the ReLU activation function than I will give here. One obvious advantage to the ReLU is that the range of the function is not squashed to values less than one. This frees the practitioner of many of the data normalization requirements typically associated with neural networks. However, the true superiority of the ReLU comes from

CONTINUED ON PAGE 38

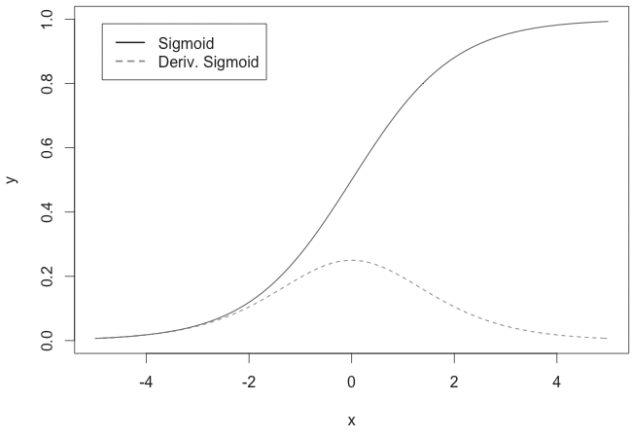
the somewhat contrived derivative of the ReLU, which is shown in Equation 3.

Equation 3: Generally Accepted Partial Derivative of the ReLU

$$\frac{dy}{dx}\phi(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

Technically the ReLU does not have a derivative at x=0; however, most neural network implementations simply treat this undefined value as 0. Figure 4 shows the derivatives of the less effective logistic and hyperbolic tangent activation functions.

FIGURE 4: SIGMOIDAL (S-SHAPED) ACTIVATION FUNCTION DERIVATIVES



The above graph shows both the logistic (sigmoid) and its derivative. The hyperbolic tangent function would look similar but shifted. The shape of the derivative indicates the problem in both cases. S-shaped activation functions saturate to zero in both directions about the x-axis. This is sometimes referred to as the vanishing gradient problem. This can cause the gradients, as calculated by the derivatives, for these neurons to drop to zero as the absolute values of these x-values become more extreme. Once the gradient for a neuron flattens to zero, the neuron will no longer train. The neural network training algorithms use this gradient to indicate what direction to move the weights.

OUTPUT ACTIVATION FUNCTIONS

Traditionally, the output layer of a neural network would use either the sigmoid, hyperbolic tangent, linear or softmax for the output activation function. Many of these choices have fallen out of favor (A. Maas, A. Hannun, A. Ng, 2014). For a regression model a linear output function should be used, for a classification model, the softmax function should be used. Never use a ReLU as the output layer activation function.

The softmax activation function is very advantageous for a classification problem. Consider a classification with five classes. Such a problem is represented by a five output neuron network. If the neural network were to output the vector [.5, 0.1, 0.75, 0.1, 0.2] you would know that the neural network had selected the third class (indicated by 0.75) as the prediction. However, 0.75 is not the probability, it is simply the largest value. The softmax activation function forces these outputs to sum to one, giving the predicted probability of the data representing each class. The softmax activation function is shown in Equation 4.

Equation 4: The Softmax Activation Function

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

Essentially you divide the natural exponent of each of the elements by the sum of all natural exponents. The value K above represents the number of output neurons present. For the vector presented above, the logloss would be [0.23 , 0.15 , 0.29 , 0.15 , 0.17]. The following URL provides a utility to calculate softmax.

http://www.heatonresearch.com/aifh/vol3/softmax.html

WEIGHT INITIALIZATION

Neural networks are initialized with random weights and biases. This creates inherently unpredictable results. This can make it very difficult to evaluate different neural network architectures to see which works best for the task at hand. While random number seeds can help produce consistent results, it is still very difficult to evaluate two different networks that have different numbers of weights. One of your candidate architectures might owe its perceived superiority more to its starting weights than the actual structure.

The Xavier weight initialization algorithm (Glorot & Bengio, 2010) has become the standard in weight initialization for neural network. This initialization samples the weights from a normal distribution with a mean of zero and a variance specified by Equation 4.

Equation 4: Xavier Weight Initialization

$$Var(W) = \frac{}{n_{in} + n_{out}}$$

The variance is equal to two divided by the sum of the number of input and output neurons for the layer. The weights resulting from

Xavier create neural networks that converge much faster than other initialization techniques. Additionally, these weight sets produce much more consistent results than many of the other weight initialization techniques.

STOCHASTIC GRADIENT DESCENT TRAINING

Stochastic Gradient Descent (SGD) with Nesterov momentum (Nesterov, 1983) has become the most commonly used training algorithm for neural networks. SGD is very similar to standard batch back propagation. Back propagation works by calculating the partial derivative of the neural network's error function for each weight. The derivatives, called gradients, are scaled by a learning rate and then added to the weights of the neural network. The gradient can be used to maximize the error of the neural network, using gradient ascent. Because we seek to minimize the error of the neural network we use the inverse of the gradient and descend to lower error levels.

Usually these changes to the weights are not applied immediately. Rather, a batch of training set elements is calculated and their gradients are summed. Once the batch is complete the weights are modified. SGD is exactly like regular batch back propagation except that a small batch size of 100-1000 elements is used. This smaller batch size is called a mini-batch. Additionally, the mini-batch is randomly sampled from the training set, with replacement. This random sampling greatly decreases overfitting.

The actual update to the weights is performed using Nesterov momentum. This is a technique that was invented by Nesterov (1983) as a general-purpose gradient descent technique. Geoffrey Hinton later recognized its value to neural network training. Nesterov momentum is a mathematically complex technique that I will not fully describe in this article. Nesterov momentum seeks to limit the damage to the weights that can be done by choosing a particularly bad mini-batch from the training elements.

CROSS ENTROPY

Neural network training algorithms have traditionally calculated error as the difference between the output neuron's actual output and expected output. This is called the quadratic error function. Research from Geoffrey Hinton has caused the quadratic error function to fall from favor. The replacement is the cross entropy function, which is shown in Equation 5.

Equation 5: Cross Entropy Error

$$CE = -\frac{1}{n}\sum_{x}[y \ln a + (1-y)\ln(1-a)]$$

In the above equation the number of training elements (n), the actual output (a) and the expected output (y) are used. The cross entropy function forces much steeper gradients for larger errors. These larger gradients cause the weights to be adjusted much faster when the error is greater and in turn causes the neural network training to converge to a lower error quicker.

L1 AND L2 REGULARIZATION

Regularization seeks to prevent overfitting by directly adjusting the weights of a neural network. The most common types of regularization are L1, L2 and dropout. The first two, L1 and L2 work by adding the neural network weights, but not the biases, to the error function. This encourages the training to keep the weights lower. This is a form of Occam's razor, in that simple weight structures are likely superior. The only differences between L1 and L2 are how they apply the weight penalty. L1 is shown in Equation 6.

Equation 6: L1 Regularization

$$E_1 = \lambda_1 \sum_{w} |w|$$

The parameter λ1 represents the relative importance of L1, a value of 1.0 means that the L1 regularization penalty is just as important

CONTINUED ON PAGE 40

as the actual error of the neural network. A value of zero turns off L1 regularization. In practice, L1 values are very low, typically less than a hundredth.

You should use L1 regularization to create sparsity in the neural network. In other words, the L1 algorithm will push many weight connections to near zero. When a weight is near zero, the program drops it from the network. Dropping weighted connections will create a sparse neural network.

Feature selection is a useful byproduct of sparse neural networks. Features are the values that the training set provides to the input neurons. Once all the weights of an input neuron reach zero, the neural network training determines that the feature is unnecessary. If your data set has a large number of input features that may not be needed, L1 regularization can help the neural network detect and ignore unnecessary features.

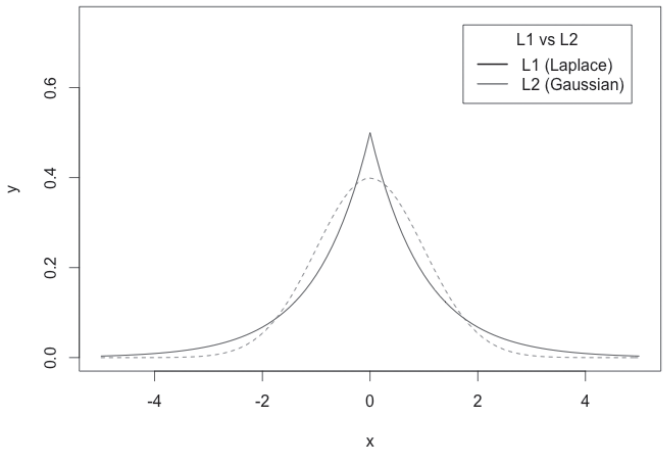
L2 regularization works similar to L1, except there is less of a focus on the removal of connections. L2 is implemented using Equation 7.

Equation 7: L2 Regularization

$$E_2 = \lambda_2 \sum_w w^2$$

The primary difference between L1 and L2 is that L1 uses the absolute value of the weights, whereas L2 uses their square. Both L1 and L2 work differently in the way that they penalize the size of a weight. L1 will force the weights into a pattern similar to a Gaussian distribution; the L2 will force the weights into a pattern similar to a Laplace distribution, as demonstrated by Figure 5.
As you can see, the L1 algorithm is more tolerant of weights further from zero, whereas the L2 algorithm is less tolerant. We will highlight other important differences between L1 and L2 in the following sections. You also need to note that both L1 and L2 count their penalties based only on weights; they do not count penalties on bias values.

FIGURE 5: L1 VS L2



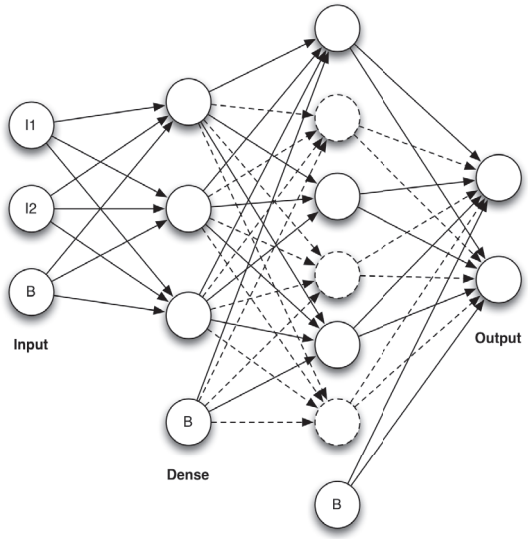
DROPOUT FOR REGULARIZATION

Hinton, Srivastava, Krizhevsky, Sutskever, & Salakhutdinov (2012) introduced the dropout regularization algorithm. Although dropout works in a different way than L1 and L2, it accomplishes the same goal—the prevention of overfitting. However, the algorithm goes about the task by actually removing neurons and connections—at least temporarily. Unlike L1 and L2, no weight penalty is added. Dropout does not directly seek to train small weights.

Most neural network frameworks implement dropout as a separate layer. Dropout layers function as a regular, densely connected neural network layer. The only difference is that the dropout layers will periodically drop some of their neurons during training. You can use dropout layers on regular feedforward neural networks. Figure 6 shows dropout in action.

The above neural network has two input neurons and two output neurons. There is also a dense and dropout layer. For each training

FIGURE 6: DROPOUT



iteration, a different set of hidden neurons is temporally dropped from the dropout layer. The dashed lines indicate the dropped neurons, and their connections. The bias neuron is never dropped. When a neuron drops out, so does its connections. Training is performed as though the dropped out neurons are not present. This forces the neural network to learn to perform even without a full

complement of neurons. The neurons become less dependent on each other.

OTHER TYPES OF NEURAL NETWORKS

It is a very exciting time for neural network research. Additional types of neural networks are actively being developed. This article focused primarily upon feedforward neural networks. However, other types of neural networks are very common. Convolutional neural networks (CNN) have become very popular for image recognition. Recurrent neural networks, particularly, gated recurrent units (GRU) have become very popular for deep time-series learning. Additionally, spiking neural networks (SNN) have found great application in the field of robotics ■

Jeff Heaton is the author of the Artificial Intelligence for Humans series of books. He is data scientist, Global R&D at RGA Reinsurance Company in Chesterfield, Mo. He can be reached at *jheaton@rgare.com.*

REFERENCES

X. Glorot & Y. Bengio. (2010). Understanding the difficulty of training deep feedforward neural networks. Journal of Machine learning Research.

K. Hornik (1991) Approximation Capabilities of Multilayer Feedforward Networks, Neural Networks, 4(2), 251–257

A. Maas, A. Hannun, A. Ng (2014). Rectifier Nonlinearities Improve Neural Network Acoustic Models

V. Nair, G. Hinton, G (2010). Rectified linear units improve restricted Boltzmann machines (PDF). ICML.

Y. Nesterov. A method of solving a convex programming problem with convergence rate O(1/sqr(k)). Soviet Mathematics Doklady, 27:372–376, 1983.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov; Dropout: A Simple Way to Prevent Neural Networks from Overfitting . Journal of Machine Learning Research (JMLR).15(Jun):1929−1958, 2014.

ENDNOTES

[1] Gibbs sampling is a Markov chain Monte Carlo (MCMC) algorithm for obtaining a sequence of observations which are approximated from a specified multivariate probability distribution. *https://en.wikipedia.org/wiki/Gibbs_sampling*