Key Takeaways

* Neural nets are a type of machine learning model that mimic biological neurons—data comes in through an input layer and flows through nodes with various activation thresholds.
* Recurrent neural networks are a type of neural net that maintain internal memory of the inputs they’ve seen before, so they can learn about time-dependent structures in streams of data.

*Machine learning has long powered many products we interact with daily–from "intelligent" assistants like Apple's Siri and Google Now, to recommendation engines like Amazon's that suggest new products to buy, to the ad ranking systems used by Google and Facebook. More recently, machine learning has entered the public consciousness because of advances in "deep learning"–these include AlphaGo's defeat of Go grandmaster Lee Sedol and impressive new products around image recognition and machine translation.*

*In this series, we'll give an introduction to some powerful but generally applicable techniques in machine learning. These include deep learning but also more traditional methods that are often all the modern business needs. After reading the articles in the series, you should have the knowledge necessary to embark on concrete machine learning experiments in a variety of areas on your own.*

*This InfoQ article is part of the series "*[*An Introduction To Machine Learning*](https://www.infoq.com/introduction-machine-learning)*". You can subscribe to receive notifications via*[*RSS*](https://www.infoq.com/feed/introduction-machine-learning)*.*

The increasing accuracy of deep neural networks for solving problems such as speech and image recognition has stoked attention and research devoted to deep learning and AI more generally. But widening popularity has also resulted in confusion. This article introduces neural networks, including brief descriptions of feed-forward neural networks and recurrent neural networks, and describes how to build a recurrent neural network that detects anomalies in time series data. To make our discussion concrete, we’ll show how to build a neural network using Deeplearning4j, a popular open-source deep-learning library for the JVM.

What are neural networks?

Artificial neural networks are algorithms initially conceived to emulate biological neurons. The analogy, however, is a loose one. The features of a biological neuron mirrored by artificial neural networks include connections between the nodes and an activation threshold, or trigger, for each neuron to fire.

By building a system of connected artificial neurons we obtain systems that can be trained to learn higher-level patterns in data and perform useful functions such as regression, classification, clustering, and prediction.

The comparison to biological neurons only goes so far. An artificial neural network is a collection of compute nodes where data represented as a numeric array is passed into a network’s input layer and proceeds through the network’s so-called hidden layers until an output or decision about the data is generated in a process described briefly below. The net’s resulting output is then compared to expected results (ground-truth labels applied to the data, for example), and the difference between the network’s guess and the right answer is used to incrementally correct the activation thresholds of the net’s nodes. As this process is repeated the net’s outputs converge on the expected results.

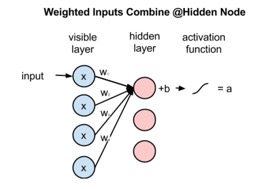
A whole neural network of many nodes can run on a single machine. It is important to note, for those coming from distributed systems, that a neural network is not necessarily a distributed system of multiple machines. Node, here, means “a place where computation occurs.”

Training process

To build a neural network, you need a basic understanding of the training process and how the net’s output is generated. While we won’t go deep into the equations, a brief description follows.

The net’s input nodes receive a numeric array, perhaps a multidimensional array called a tensor, that represents the input data. For example, each pixel in an image may be represented by a scalar that is then fed to a node. That input data passes through the coefficients, or parameters, of the net and through multiplication those coefficients will amplify or mute the input, depending on its learned importance—that is, whether or not that pixel should affect the net’s decision about the entire input.

Initially, the coefficients are random; that is, the network is created knowing nothing about the structure of the data. The activation function of each node determines the output of that node given an input or set of inputs. So the node fires or does not, depending on whether the strength of the stimulus it receives, the product of the input and the coefficient, surpasses the threshold of activation.



In a so-called dense or fully connected layer, the output of each node is passed to all nodes of the subsequent layer. This continues through all hidden dense layers to end with the output layer, where a decision about the input is reached. At the output layer, the net’s decision about the input is evaluated against the expected decision (e.g., do the pixels in this image represent a cat or a dog?). The error is calculated by comparing the net’s guess to the true answer contained in a test set, and using that error, the coefficients of the network are updated in order to change how the net assigns importance to different pixels in the image. The goal is to decrease the error between generated and expected outputs—to correctly label a dog as a dog.

While deep learning is a complicated process involving matrix algebra, derivatives, probability and intensive hardware utilization as large matrices of coefficients are modified, the end user does not need to be exposed to all the complexity.

There are, however, some basic parameters that you should be aware of that will help you understand how neural networks function. These are the activation function, optimization algorithm, and objective function (also known as the loss, cost or error function).

The activation function determines whether and to what extent a signal should be sent to connected nodes. A frequently used activation is just a basic step function that is 0 if its input is less than some threshold and 1 if its input is greater than the threshold. A node with a step-function activation function thus either sends a 0 or 1 to connected nodes. The optimization algorithm determines how the network learns, and more accurately how weights are modified after determining the error. The most common optimization algorithm used is [stochastic gradient descent](https://en.wikipedia.org/wiki/Stochastic_gradient_descent). Finally, a cost function is a measure of error, which gauges how well the neural network performed when making decisions about a given training sample, compared to the expected output.

Open-source frameworks such as [Keras](https://keras.io/) for Python or [Deeplearning4j](https://deeplearning4j.org/) for the JVM make it fairly easy to get started building neural networks. Deciding on what network architecture to use often involves matching your data type to a known solved problem and then modifying an existing architecture to suit your use case.

Types of neural networks and their applications

Neural networks have been known and used for many decades. But a number of important technological trends have recently made deep neural nets much more effective.

Computing power has increased with the advent of GPUs to increase the speed of the matrix operations, as well as with larger distributed computing frameworks, making it possible to train neural nets faster and iterate quickly through many combinations of hyperparameters to find the right architecture.

Larger datasets are being generated, and large high-quality labeled datasets such as ImageNet have been created. As a rule, the more data a machine learning algorithm is trained on, the more accurate it will be.

Finally, advances in how we understand and build the neural network algorithms have resulted in neural networks consistently setting new accuracy records in competitions for computer vision, speech recognition, machine translation and many other machine perception and goal-oriented tasks.

Although the universe of neural network architectures is very large, the main types of networks that have seen wide use are described here.

Feed-forward neural networks

This is a neural network that has an input layer, an output layer, and one or more hidden layers. Feed forward neural networks make good universal approximators (functions that map any input to any output) and can be used to build general-purpose models.

This type of neural network can be used for both classification and regression. For example, when using a feed-forward network for classification the number of neurons on the output layer is equal to the number of classes. Conceptually, the output neuron that fires determines the class that the network has predicted. More accurately, each output neuron returns a probability that the record matches that class, and the class with the highest probability is chosen as the model’s output classification.

The benefit of feed-forward neural networks such as multilayer perceptrons is that are the easy to use, less complicated than other types of nets and a wide variety of examples are available.

Convolutional neural networks

Convolutional neural networks are similar to feed-forward neural nets, at least in the way that data passes through the network. Their form is roughly modeled on the visual cortex. [Convolutional nets](https://deeplearning4j.org/convolutionalnets) pass several filters like magnifying glasses over an underlying image. Those filters focus on feature recognition on a subset of the image, a patch or tile, and then repeat that process in a series of tiles across the image field. Each filter is looking for a different pattern in the visual data; for example, one might look for a horizontal line, another might look for a diagonal line, another for a vertical. Those lines are known as features and the as the filters pass over the image, they construct feature maps locating each kind of line each time it occurs at different places in the image. Different objects in images -- cats, 747s, masticating juicers -- generate different sorts of features maps, which can ultimately be used to classify photos. Convolutional networks have proven very useful in the field of image and video recognition (and because sound can be represented visually in the form of a spectrogram, convolutional networks are widely used for voice recognition and machine transcription tasks as well).

Convolutional vs. Feed Forward Nets for image processing. Both network types can analyze images, but how they analyze them is different. While a convolutional neural network steps through overlapping sections of the image and trains by learning to recognize features in each section, a feed forward network trains on the complete image. A feed forward network trained on images where a feature is always in a particular position or orientation may not recognize when that feature shows up in an uncommon position, while a convolutional network would if trained well.

Convolutional neural networks are used for tasks such as image, video, voice, and sound recognition as well as autonomous vehicles.

This article focuses on recurrent neural networks, but convolutional neural networks have performed so well with image recognition that we should acknowledge their utility.

Recurrent neural networks (RNNs)

Unlike feed-forward neural networks, the hidden layer nodes of a recurrent neural network maintain an internal state, a memory, that is updated with new input the network is fed. Those nodes make decisions both based on the current input and also what has come before. Recurrent neural networks can make use that internal state to process relevant data in arbitrary sequences of inputs, such as time series.

They are used for handwriting recognition, speech recognition, log analysis, fraud detection, cybersecurity.

Recurrent neural nets are best for datasets that contain a temporal dimension, like logs of web or server activity; sensor data from hardware or medical devices; financial transactions; or call records. Tracking dependencies and correlations within data over many time steps requires that you know the current state and some number of previous states. Although this might be possible with a typical feed-forward network that ingests a window of events, and subsequently moves that window through time, such an approach would limit us to dependencies captured by the window, and the solution would not be flexible.

A better approach to track long-term dependencies over time is some sort of "memory" that stores significant events so that later events can be understood and classified in context. The beauty of an RNN is that the "memory" contained in its hidden layers learns the significance of these time-dependent features on its own over very long windows.

In what follows, we will discuss the application of recurrent networks to both character generation and network anomaly detection. What makes an RNN useful for anomaly detection in time series data is this ability to detect dependent features across many time steps.

Applying recurrent neural networks

Although our example will be monitoring activity on a computer network, it might be useful to  start by discussing a simpler use case for RNNs. The internet has multiple examples of using RNNs for generating text, one character at a time, after being trained on a corpus of text to predict the next letter given what’s gone before. Let’s take a look at the features of an RNN by looking more closely at that use case.

RNNs for character generation

Recurrent neural networks can be trained to treat characters in the English language as a series of time-dependent events. The network will learn that one character frequently follows another (“e” follows “h” in “the”, “he,” and “she”) and as it makes predictions over the next character in the sequence, it will be trained to reduce error by comparisons with actual English text.

When fed the complete works of Shakespeare, it can then generate impressively Shakespeare-like output; [for example](http://karpathy.github.io/2015/05/21/rnn-effectiveness/), “Why, Salisbury must find his flesh and thought…”. When Fed a sufficiently large amount of Java code, it will emit something that almost compiles.

Java is an interesting example because its structure includes many nested dependencies. Every parenthesis that is opened will eventually be closed. Every open curly brace has a closed curly  brace down the line. These are dependencies not located immediately next to one another—the distance between multiple events can vary. Without being told about these dependent structures, a recurrent neural net will learn them.

In anomaly detection, we will be asking our neural net to learn similar, perhaps hidden or non-obvious patterns in data. Just as a character generator understands the structure of data well enough to generate a simulacrum of it, an RNN used for anomaly detection understands the structure of the data well enough to know whether what it is fed looks normal, or not...

The character generation example is useful to show that RNN's are capable of learning temporal dependencies over varying ranges of time. An RNN can use that same capability for anomaly detection in network activity logs.

Applied to text, anomaly detection might surface grammatical errors, because grammar  structures what we write. Likewise, network behavior has a structure; it follows predictable patterns that can be learned. An RNN trained on normal network activity would perceive a network intrusion to be as anomalous as a sentence without punctuation

A sample network anomaly detection project

Suppose we wanted to detect network anomalies with the understanding that an anomaly might point to hardware failure, application failure, or an intrusion.

What our model will show us

The RNN will train on a numeric representation of network activity logs, feature vectors that translate the raw mix of text and numerical data in logs.

By feeding a large volume of network activity logs, with each log line a time step, to the RNN, the neural net will learn what normal expected network activity looks like. When this trained network is fed new activity from the network, it will be able to classify the activity as normal and expected, or anomalous.

Training a neural net to recognize expected behavior has an advantage, because it is rare to have a large volume of abnormal data, or certainly not enough to accurately classify all abnormal behavior. We train our network on the normal data we have, so that it alerts us to non-normal activity in the future. We train for the opposite where we have enough data about attacks.

As an aside, the trained network does not necessarily note that certain activities happen at certain times (it does not know that a particular day is Sunday), but it does notice those more obvious temporal patterns we would be aware of, along with other connections between events that might not be apparent.

We’ll outline how to approach this problem using Deeplearning4j, a widely used open-source library for deep learning on the JVM. Deeplearning4j comes with a variety of tools that are  useful throughout the model development process: DataVec is a collection of tools to assist with the extract-transform-load (ETL) tasks used to prepare data for model training. Just as Sqoop helps load data into Hadoop, DataVec helps load data into neural nets by cleaning, preprocessing, normalizing and standardizing data. It’s similar to Trifacta’s Wrangler but focused a bit more on binary data.

Getting started

The first stage includes typical big data tasks and ETL: We need to gather, move, store, prepare, normalize, and vectorize the logs. The size of the time steps must be decided. Data transformation may require significant effort, since JSON logs, text logs, and logs with  inconsistent labeling patterns will have to be read and converted into a numeric array.  DataVec can help transform and normalize that data. As is the norm when developing machine learning models, the data must be split into a training set and a test (or evaluation) set.

Training the network

The net’s initial training will run on the training split of the input data.

For the first training runs, you may need to adjust some hyperparameters (“hyperparameters” are parameters that control the “configuration” of the model and how it trains) so that the model actually learns from the data, and does so in a reasonable amount of time. We discuss a few hyperparameters below. As the model trains, you should look for a steady decrease in error.

There is a risk that a neural network model will "overfit" on the data. A model that has been trained to the point of overfitting the dataset will get good scores on the training data, but will not make accurate decisions about data it has never seen before. It doesn’t “generalize” -- in machine-learning parlance. Deeplearning4J provides regularization tools and “early stopping” that help prevent overfitting while training.

Training the neural net is the step that will take the most time and hardware. Running training on GPUs will lead to a significant decrease in training time, especially for image recognition, but additional hardware comes with additional cost, so it’s important that your deep-learning framework use hardware as efficiently as possible. Cloud services such as Azure and Amazon provide access to GPU-based instances, and neural nets can be trained on heterogenous clusters with scalable commodity servers as well as purpose-built machines.

Productionizing the model

Deeplearning4J provides a ModelSerializer class to save a trained model. A trained model can  be saved and either be used (i.e., deployed to production) or updated later with further training.

When performing network anomaly detection in production, log files need to be serialized into the same format that the model trained on, and based on the output of the neural network, you would get reports on whether the current activity was in the range of normal expected network behavior.

Sample code

The configuration of a recurrent neural network might look something like this:

MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()

.seed(123)

.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC\_GRADIENT\_DESCENT).iterations(1)

.weightInit(WeightInit.XAVIER)

.updater(Updater.NESTEROVS).momentum(0.9)

.learningRate(0.005)

.gradientNormalization(GradientNormalization.ClipElementWiseAbsoluteValue)

.gradientNormalizationThreshold(0.5)

.list()

.layer(0, new GravesLSTM.Builder().activation("tanh").nIn(1).nOut(10).build())

.layer(1, new RnnOutputLayer.Builder(LossFunctions.LossFunction.MCXENT)

.activation("softmax").nIn(10).nOut(numLabelClasses).build())

.pretrain(false).backprop(true).build();

MultiLayerNetwork net = new MultiLayerNetwork(conf);

net.init();

Let’s describe a few important lines of this code:

* .seed(123)

sets a random seed to initialize the neural net’s weights, in order to obtain reproducible results. Typically, coefficients are initialized randomly, and so to obtain consistent results while adjusting other hyperparameters, we need to set a seed, so we can use the same random weights over and over as we tune and test.

* .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC\_GRADIENT\_DESCENT).iterations(1)

determines which optimization algorithm to use (in this case, stochastic gradient descent) to determine how to modify the weights to improve the error score. You probably won’t have to modify this.

* .learningRate(0.005)

When using stochastic gradient descent, the error gradient (that is, the relation of a change in coefficients to a change in the net’s error) is calculated and the weights are moved along this gradient in an attempt to move the error towards a minimum.  SGD gives us the direction of less error, and the learning rate determines how big of a step is taken in that direction. If the learning rate is too high, you may overshoot the error minimum; if it is too low, your training will take forever. This is a hyperparameter you may need to adjust.

Getting Help

There is an active community of Deeplearning4J users who can be found on several support channels on [Gitter](https://gitter.im/deeplearning4j/deeplearning4j).

About the author

**Tom Hanlon** is currently at Skymind.IO where he is developing a Training Program for Deeplearning4J. The consistent thread in Tom’s career has been data, from MySQL to Hadoop and now neural networks.

*Machine learning has long powered many products we interact with daily–from "intelligent" assistants like Apple's Siri and Google Now, to recommendation engines like Amazon's that suggest new products to buy, to the ad ranking systems used by Google and Facebook. More recently, machine learning has entered the public consciousness because of advances in "deep learning"–these include AlphaGo's defeat of Go grandmaster Lee Sedol and impressive new products around image recognition and machine translation.*

*In this series, we'll give an introduction to some powerful but generally applicable techniques in machine learning. These include deep learning but also more traditional methods that are often all the modern business needs. After reading the articles in the series, you should have the knowledge necessary to embark on concrete machine learning experiments in a variety of areas on your own.*