Neural Networks of Arbitrary Topology

ALEX SHADLEY
University of Kansas
December 7, 2017

Introduction

Artificial Neural Networks (ANNs) are a large and rapidly developing area of machine learning research. As we continue to find new applications for neural networks, the demand for specialized forms of neural networks more optimized for handling specific tasks grows. While more traditionally constructed networks, such as fully connected networks, are robust in many different tasks, they have been consistently outperformed in certain areas by specialized networks, for example convolutional neural networks with image classification [2]. Alternatives to fully connected layers can also result in improved performance while maintaining explanatory power and depth of analysis [1]. We aim to produce a neural network model capable of encompassing many different forms of specialty network, such that these networks may be better understood and more easily prototyped.

Artificial Neurons

Advanced implementations of neural networks involve complex network topologies capable of performing considerable feats of generalization and reasoning. However, since these networks are composed of individual building blocks, known as artificial neurons, it is crucial to first understand these components. Furthermore, these individual neurons can perform useful tasks, such as classification, on their own. In this section we explore two such components, the Perceptron and the Adaline.

Perceptrons

The Perceptron was first introduced by Rosenblatt in the late 1950's [3] as a simple device that could be used to learn basic patterns in data, and then to

recognize these patterns when presented with similar data. The first Perceptrons were conceived as physical devices composed entirely of hardware, but due to considerable improvements in computing, they can now be easily implemented entirely with software. Software implementations are highly preferable in many cases, due to their extensibility and scalability.

From a purely mathematical perspective, a Perceptron is a self-contained unit that takes a vector of inputs and produces an output. We often speak of an artificial neuron 'activating', by which we mean outputting a positive value in response to an input. This output is dependent both on the inputs to the Perceptron, $x \in \mathbb{R}^n$, and on the weights of the Perceptron, $w \in \mathbb{R}^n$. The output is also dependent on a threshold weight, θ. w is a vector of the same cardinality as x, and is an integral part of the Perceptron model. w represents the Perceptron's learned knowledge of a pattern, and dictates how it classifies future inputs according to this pattern. Each weight, or element of w, corresponds to an element of x. When the Perceptron evaluates an input x, it multiplies each element of x by its corresponding weight in w, then sums these products together. This is the *weighted sum*, represented as *net*. Since x and w are vectors, *net* can be described with the dot product,

$$net = x \cdot w + \theta, \tag{1}$$

where θ is the threshold weight, an additional weight that acts independently of an input. θ can be thought of as defining the threshold at which the neuron will activate. After the weighted sum is calculated, it is passed through an activation function $F(s)$ to obtain the output of the Perceptron. Thus, the output y of the Perceptron can be defined as

$$y(x) = F(x \cdot w + \theta) = F(net) \tag{2}$$

Note that *net* is a mostly superfluous intermediary for now, but it becomes useful in defining partial derivatives later. In the case of the Perceptron, $F(s)$ is a slight modification of the sign function, specifically

$$F(s) = \begin{cases} -1 & s \leq 0 \\ 1 & s > 0 \end{cases} \tag{3}$$

Training

Training is the process by which the weights of the Perceptron are updated to more accurately detect a pattern. This is accomplished with a set of training data D, where $D = \{(X_1, l_1), (X_2, l_2), ..., (X_n, l_n)\}$. Here X is a vector of input values fed to the Perceptron, and l is the *label* of the training example.

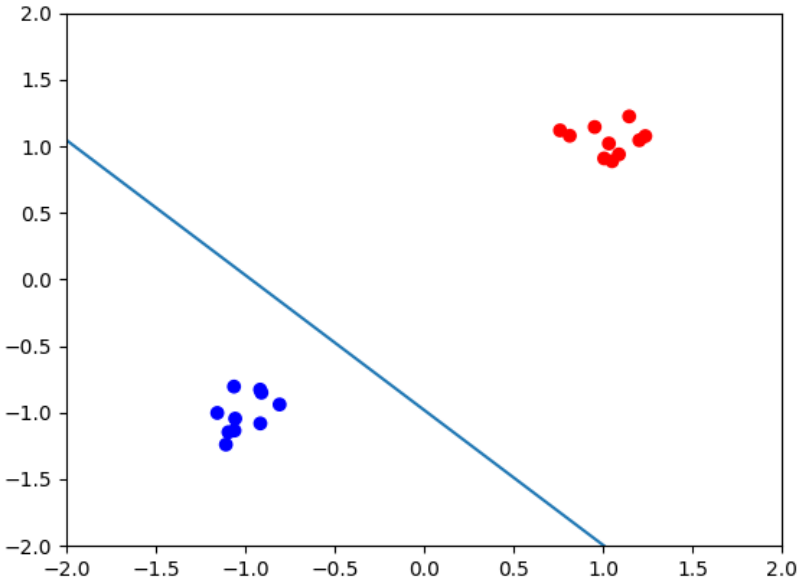
The label is the expected output of the Perceptron, given X as an input. From this perspective, a single training example is notated as $d \in D$.

Learning is accomplished with a simple algorithm. Each element in a set of data is classified by the Perceptron, and for each incorrect classification, the weights are updated according to the rule:

$$w_j \leftarrow w_j + (l - y(X)) \cdot X_j \cdot \gamma \tag{4}$$

This defines the update rule for a given element d in the training set of data, for a specific weight w_j, and its associated input from the training example, X_j. γ is the learning rate, usually a very small value used to moderate the learning process. This rule would be repeated over each weight and over each training element to achieve desirable results.

In practice, the Perceptron is indeed capable of classifying sets of data, shown in the image below.



Adaline

While the Perceptron model is useful for binary sets of linearly separable data, where the label applied to each set of inputs can only hold one of two values, it fails to model sets of continuous data. This is where the Adaline comes in. The Adaline is very similar to the Perceptron, with one major difference being the activation function. Instead of a stepwise activation

function, Adalines use a continuously valued function, examples being linear and sigmoid.

Activation Function

A number of options exist for a continuously valued activation function. A simple approach is to use a linear activation function, which is to say, not modify the output in any way. With this activation function, the output of the neuron would be:

$$y(x) = x \cdot w + \theta \tag{5}$$

While this method is simple, it is less valuable for classification tasks, since with classification we only deal with points having discrete values of 1 and -1. In these cases, it is useful to have a function bounded at -1 and 1. A class of functions known as sigmoids fit this description nicely, so we use one such function, tanh, which is defined as follows.

$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{6}$$

Gradient Descent

Another important development is that the Adaline model uses gradient descent to minimize error and learn the training set of data. Gradient descent makes use of the continuous activation function, in this case σ, as defined above.

$$E_d(\theta, w) = \frac{1}{2}(\sigma(\theta + w \cdot X_d) - l_d)^2 \tag{7}$$

This defines the error E_d for a single training example $d \in D$, where X_d are the inputs of d, and l_d represents the label, or desired output. Furthermore, the total error of the neural network over the entire training set D can be found by summing each individual error.

$$E = \sum_{d \in D} E_d(\theta, w) \tag{8}$$

Thus, our problem of training weights can be defined as an optimization problem, where we try to minimize E. In order to perform gradient descent on the error function, we must first find its derivative. First note that since we seek to use an iterative algorithm to update weights, it is more useful

to define the derivative of E_d. We then compute the partial derivative with respect to each weight in w.

$$\begin{aligned}\frac{\partial E_d}{\partial w_j} &= \frac{\partial}{\partial w_j}\Big[\frac{1}{2}(\sigma(\theta + w \cdot X_d) - l_d)^2\Big] \\ &= (\sigma(\theta + w \cdot X_d) - l_d)(\sigma'(\theta + w \cdot X_d)) \cdot (X_d)_j\end{aligned} \tag{9}$$

Observe that w_j refers to the j-th element of the weights matrix w, and that $(X_d)_j$ refers to the corresponding j-th element of X_d. Using this definition, we can then obtain a weight adjustment to use in our algorithm.

$$\delta_d = (l_d - \sigma(\theta + w \cdot X_d))(\sigma'(\theta + w \cdot X_d)) \tag{10}$$

Thus, our update rule becomes:

$$w_j \leftarrow w_j + \gamma \cdot \delta_d \cdot (X_d)_j \tag{11}$$

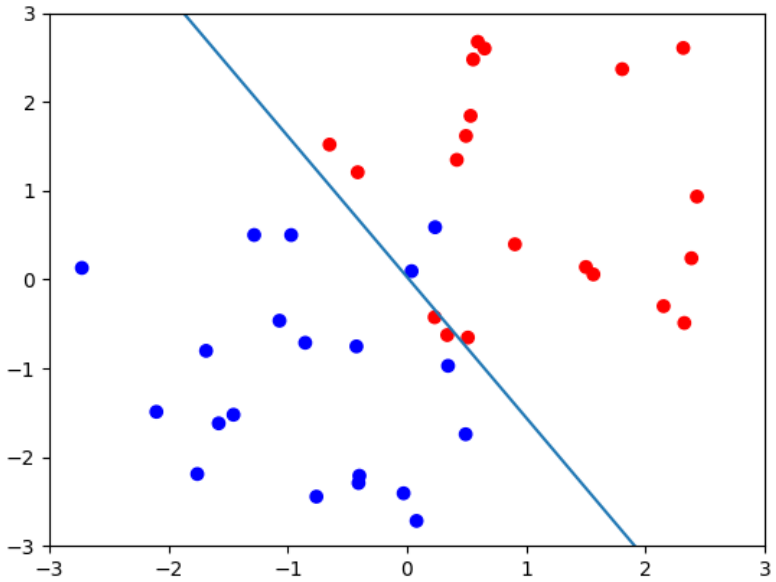
With this update rule, we design an algorithm to train our weights to a particular set of training data.

1. Select a member from our set of data, either sequentially or randomly.
2. Execute the update rule for each weight in w.
3. Repeat this process over the entire set of training data.

This process can be repeated multiple times over the same training data to train weights further if desired.

Advantages

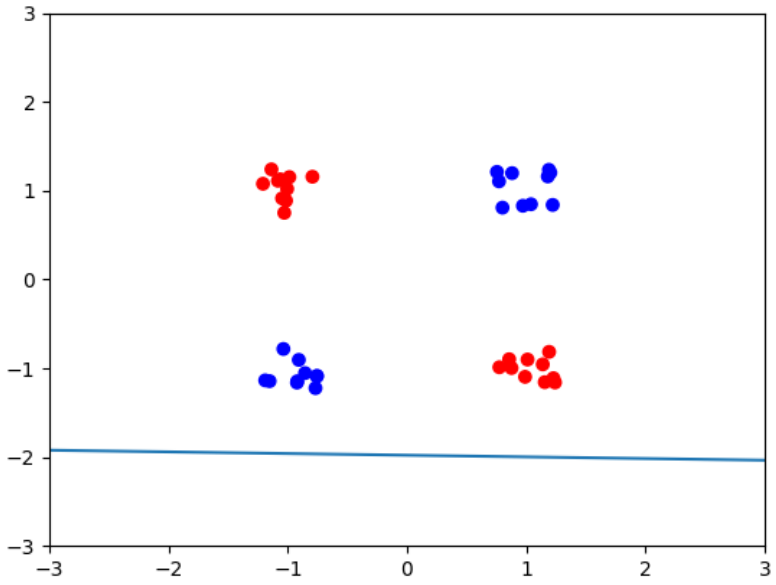
As discussed above, one of the major advantages of Adalines over Perceptrons is the ability to process continuously valued data, as opposed to sets labeled at -1 and 1. Another major advantage of the Adaline model is its robustness when faced with messy data. In such sets, where linear separation is not possible, a useful if not completely accurate classification can be performed. The diagram below demonstrates an Adaline attempting to classify one such messy set.



Limitations

While single neurons are somewhat useful in solving a limited class of problems, many problems, such as classifications with non-linearly separable data, are impossible for these simple systems. The primary weakness of the single layer approach is that these models are inherently limited to linear separations. If data is not linearly separable, meaning that a line separating two differently labeled sets does not exist, single layer networks cannot produce an effective solution.

This issue is perhaps best demonstrated by what is known as the *xor problem*:



This is referred to as the xor problem since both inputs being positive and both inputs being negative are associated with one classification, and one positive and the other negative associated with another classification. This closely mirrors the behavior of the standard xor bitwise operator.

Clearly, there is no linear rule that we can use to classify these two sets of data, and thus is unsolvable with the Perceptron and Adaline.

Feed-Forward Networks

To overcome the limitations of single-neuron systems such as the Perceptron, neurons can instead be linked together to form more complex networks, capable of much more advanced reasoning and able to approximate much more complex functions. These networks are loosely motivated by biological learning systems. These are understood to consist of many small units, Neurons, linked together to produce a biological network capable of complex reasoning. Artificial neural networks are built on the same principle, although it is important to note that we seek to use neural networks only as a computational model, rather than to simulate biological processes.

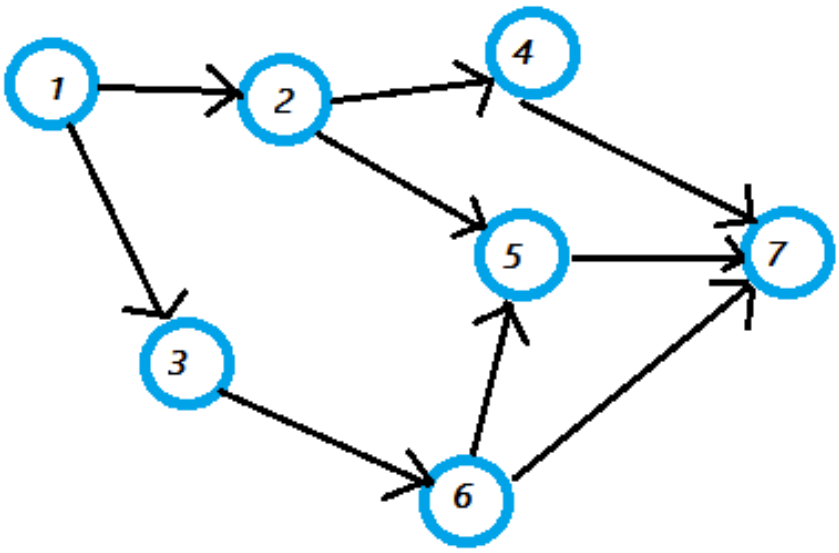
We also limit ourselves here to feed-forward networks, as opposed to recurrent networks that allow for loops in the neuron topology. These networks are advantageous in that they can retain a sort of memory, but they are not the focus of this paper.

Feed-Forward Network Topologies

The topology of a network describes how the neurons that make it up interact with one another. Since neural networks are a network-based computational model, it follows that a graph is a convenient way to model this topology. Here we take a graph to mean a set of points, or *vertices*, connected by lines, or *edges*. In these representations, a vertex represents a neuron and an edge represents information flow from one neuron to another. The information transfers here are the output of one neuron acting as an input of another. Since information is only ever transmitted in one direction from one neuron to another, we draw these edges as being directed from the neuron sending information to the neuron receiving information, and thus our graph is a directed graph or *digraph*. A digraph is a graph with the additional stipulation that each edge must have a direction from one of the vertices it connects to the other. Defining our neural network in terms of a digraph is very useful, since this is inherently a very flexible model, and we may wish to describe many different types of network.

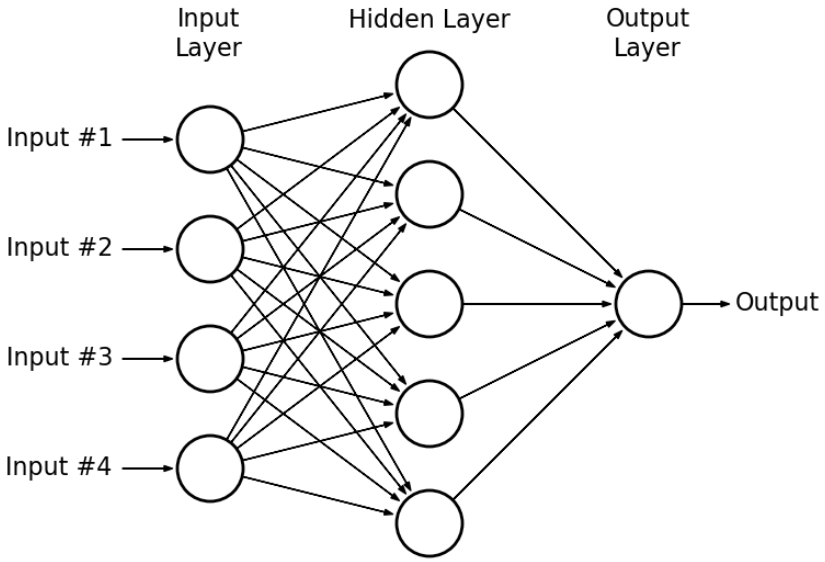
While we do want to maintain the flexibility of our model, a few restraints must be imposed to ensure a workable network. The most important of these is that the digraph must be *acyclic*, meaning that it must not contain any directed loops. This is a constraint common to all feed-forward networks, since a directed loop would result in a closed chain of neuron inputs that would be irresolvable.

Another way to think about directed acyclic graphs is by imagining traversing the graph from an arbitrarily chosen starting point. We move from one vertex to the next, along edges and only in the direction of these edges. No matter which starting point we pick, we will never be able to return to this same point during our traversal. Furthermore, if we continue our traversal for long enough, and assuming of course that our graph is finite, we will eventually arrive at a vertex with no out-edges, and thus be unable to leave that vertex. An example of a directed acyclic graph is shown below.



Layered Models

Many models call for an additional restriction of layers to be imposed. Layers here are defined as a set of vertices, unrelated to each other in the set. Layer-based models typically consist of a specifically ordered sequence of layers, where connections exist only between vertices of adjacent layers, and where the direction of these connections is always from the preceding layer to the following layer. Applying this concept to neural networks, we can see layers as containing sets of neurons. Moreover, each neuron's inputs should consist entirely of outputs of neurons belonging to the previous layer. If every possible connection is made between all layers, then the network is considered *fully connected*. This image demonstrates a typical feed-forward, fully connected network with three layers:



While the constraint of layers is certainly useful from an organizational standpoint, some flexibility is certainly lost. Some applications call for connections within a layer, and others call for connections between non-adjacent layers. If enough of these mutations are employed, it becomes clear that the abstraction of layers is no longer accurate or helpful. It is important to remember that, as long as a network is acyclic, it is a valid feed-forward network, and can be simulated as such.

Neuron-Centric Model

Due to the limitations imposed by a layered model, we propose a new model that completely does away with the restriction of layers, instead allowing any network topology that conforms to a directed acyclic graph. This model instead focuses on the computations performed at the neuron level, and on the interactions between individual neurons. While this is a shift in perspective, all of the theory behind feed-forward neural networks still applies to the neuron-centric model. Also note that it is still possible to produce layered neural networks with this model.

Upstream and Downstream

To better discuss the intricacies of these systems, we introduce some additional terminology. For any given vertex v, we will refer to the set of *upstream* vertices with respect to v, $upstream(v)$, as the vertices directly con-

nected to v in the direction of v. Likewise, the set of downstream vertices with respect to v, also represented as $downstream(v)$, consists of the vertices directly connected to v where these connections are directed away from v.

As these concepts apply in neural networks, the upstream neurons for any neuron n will be the inputs of n, and the downstream neurons of n are the neurons that use the output of n as an input.

Neuron Types

The neuron-centric model recognizes 3 distinct types of neuron:

1. Input
2. Hidden
3. Output

These neuron types are analogous to the layer types seen in layered models. Input neurons do not perform any computations or posses any weights, but rather receive an input to the network and pass this value to all downstream neurons. Input neurons can never have any upstream neurons (i.e. its only input will be from outside the network), and must have at least 1 downstream neuron to have any impact on the network. Thus the output of an input is given by:

$$y(x) = x. \tag{12}$$

Hidden neurons behave exactly as neurons in any other system, taking inputs from upstream neurons (either input or hidden) and producing an output that is then sent to downstream neurons. Output neurons behave very similarly to hidden neurons, with the caveat that instead of their output becoming the input of a neuron, it instead acts as a network output. Output neurons can thus never have any downstream neurons. Output and hidden neurons evaluate their outputs with the same equation:

$$y(x) = \sigma(x \cdot w + \theta), \tag{13}$$

where σ is the activation function, x is the vector of inputs and w the corresponding vector of weights. We use tanh as our activation function, which is bounded at -1 and 1, but any standard activation function would be suitable.

Evaluation

Evaluation is the feed-forward process by which the network produces outputs from a set of inputs. This is done by feeding each input to an input neuron, then calculating the output y of each output neuron k.

$$y(x) = \sigma(x \cdot w + \theta) \tag{14}$$

Note that x, the vector of inputs to k, consists of the outputs of other neurons. By substituting in the output function for the neuron of each input to k, we can extend the equation backwards. If we repeat this process, we will eventually end up with only input neurons, which have known values, and thus be able to compute the output of the network. Evaluating neural networks in the neuron-centric model relies on this recursive method, as it accurately describes any acyclic network.

Learning

The task of training a neural network on an input involves three essential steps:

1. Feeding the inputs forward and obtaining the resulting output
2. Back-propagating the error in the network to all neurons
3. Updating the weights of each neuron based on the error in each neuron

Error

Optimizing the weights of a complex network may seem like a daunting task at first, but by defining an error function for our network and then trying to minimize this function, we have posed the problem of training weights as an optimization problem, for which we will use gradient descent. Note that the error, E, is a function of all weights in the network, and that by performing gradient descent on this function we can converge onto the optimal values for each weight.

$$E_{kd} = \frac{1}{2}(l_{kd} - y_{kd})^2 \tag{15}$$

Note that this only gives the error for some training example d and for some output neuron k, where l_{kd} is the target output, or label, of the network for d, and y_{kd} is the actual output of the network. If instead we want the error of the entire network for some training example d, we get:

$$E_d = \frac{1}{2} \sum_{k \in outputs} (l_{kd} - y_{kd})^2 \tag{16}$$

Backpropagation

Backpropagation is the second step of the learning process, where error is propagated backwards through the network and weights are updated. As mentioned above, this is posed as an optimization problem, where we use the gradient descent algorithm to minimize E_d.

At this point we introduce some more specific notation to deal with specific neurons in a network. For a neuron j, x_{ji} refers to the input from i to j, and w_{ji} refers to the corresponding weight. $Downstream(j)$ is the set of all downstream neurons of j.

Gradient descent requires finding the partial derivative of error with respect to each weight, $\frac{\partial E_d}{\partial w_{ji}}$. The chain rule then gives us

$$\begin{aligned} \frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial net_j} x_{ji} \end{aligned} \tag{17}$$

The problem has then been reduced to finding the partial of error with respect to net_j.

$$\begin{aligned} \frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial y_j} \frac{\partial y_j}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k w_{kj} \sigma'(net_j) \end{aligned} \tag{18}$$

Then, using δ_j to represent $\frac{\partial E_d}{\partial net_j}$, we get

$$\delta_j = \sigma'(net_j) \sum_{k \in Downstream(j)} \delta_k \dot{w}_{k_j} \tag{19}$$

Given this definition of δ_j, the update rule for w_{ji} becomes:

$$w_{ji} \leftarrow w_{ji} + \gamma \delta_j x_{ji} \tag{20}$$

In this equation, γ is our learning rate, usually a small value such as .1

Implementation

One of the main priorities with our implementation is to achieve the simplest and most efficient solution possible, while still preserving as much flexibility and modularity as possible. To achieve this, Object Oriented Programming (OOP) techniques are used to encapsulate parameters and processes. The implementation makes most use of two classes, Network and Neuron. Neuron carries out the function of a single neuron, taking inputs, calculating outputs, calculating error, and adjusting weights accordingly. Network is the overarching class used to construct and activate the network. A key implementation choice is the emphasis on Neuron-centric function – almost all of the actual execution of the network's function is carried out at the Neuron level. In this sense, the Network class can be thought of as a wrapper or level of abstraction surrounding our neural network.

As discussed above, while the restriction of layers is a common convention, it is by no means integral to the function of a feed-forward network, and because of this, our implementation of a neural network eschews any notion of a layer.

Neuron

At all times, Neuron will keep track of all of its upstream and downstream neurons. Each neuron must know what its upstream neurons are to receive input during feed-forward operations, and likewise must know its downstream neurons since its error is based on downstream neurons. The neuron also keeps track of its weights, and can update them accordingly during backpropagation.

During both the feed-forward and backpropogation processes, the Neuron class takes advantage of recursion to accurately simulate the decentralized and non-uniformly constructed network that it makes up. At the Neuron level, the process for querying the Neuron's output works as follows:

1. When the Neuron is queried for its output, it first checks to see if it has already calculated this value from a past query, and returns this value if so.
2. If the output has not been calculated, the Neuron queries each of its upstream neurons for their respective inputs.
3. After receiving all inputs, the Neuron then calculates its output using these inputs, its weights, and its activation function, and finally returns this output.

This approach closely mimics the recursive mathematics that inspire it. While one single Neuron is responsible for very little, it can be seen that querying a Neuron kicks off a recursive process wherein the first Neuron queries its upstream neurons, which in turn query their upstream neurons, and on until the calculation reaches the input neurons of the network. This recursive process makes up the feed-forward process.

Backpropogation occurs through a very similar mechanism. The differences are in this case that neurons are queried for their delta (a concept roughly equivalent to 'the error in one single Neuron'), and neurons query their downstream neurons instead of upstream.

Network

The Network class serves two primary functions. First, it constructs the network, which entails initializing each neuron and setting all of the connections between neurons. Second, it activates the network and returns the results when its functions **evaluate** and **learn** are called. This distilled version of a network is critical to its modularity and extensibility. Many more complex network topologies can be implemented using this class as a basis.

When evaluating the neural network for a set of inputs, the Network class first sets all of its input neurons accordingly to the inputs it was passed. Then, the Network initiates the feed-forward process by querying each output neuron for its output. By querying the output neurons, the recursive evaluation is begun, ensuring that each neuron calculates its proper output.

When learning a training example, the Network first evaluates the inputs with the recursive feed-forward evaluation. The output of this evaluation is then compared to the training example's label, and the error is backpropogated in a similar manner to the feed-forward method.

Applications

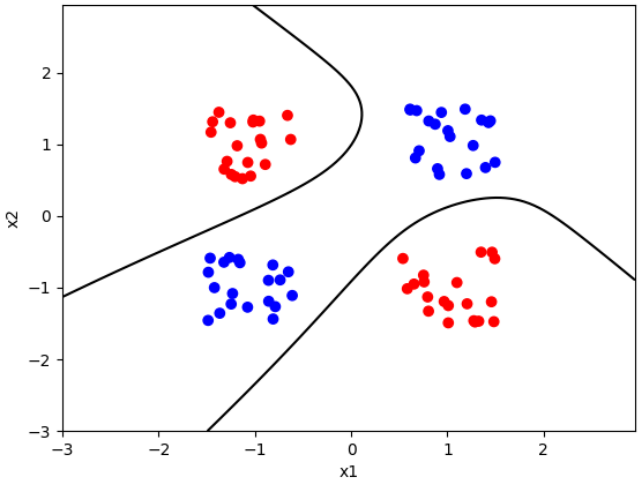
Neural networks have a very promising variety and depth of applications. Due to their incredible flexibility and capacity to universally approximate functions, neural networks can be used to solve many problems across many fields. Sometimes, the more generalized fully-connected network can be applied directly to a problem, but often specialized forms of neural networks can be designed to more specifically meet the needs of a problem. Here we discuss two such applications, time series analysis and image classification.

Classification With Simulated Data

To test our learning algorithm, we can generate an easily visualized classification problems using two inputs, x_1 and x_2. Training examples belonging to one category will have an expected output of 1, and examples belonging to the other category will have an expected output of -1.

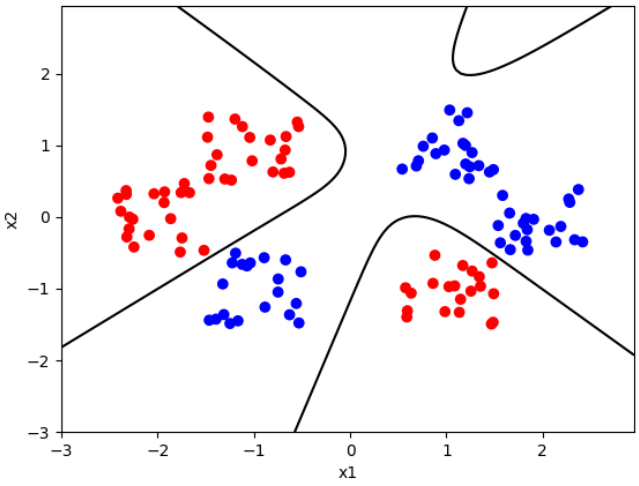
Since we have 2 inputs, the network should have 2 inputs neurons, and with 2 possible classifications, only 1 output neuron is needed. The number of hidden neurons required depends on the complexity of the pattern to be fit, and 4 is used for these results.

This is a visualization of a classification learned by the described network on the xor problem:



Here, the red dots represent training examples in the positive category (expected output = 1) and blue dots represent examples in the negative category (expected output = -1). The black lines shown are the contour of the function produced by the network where $f = 0$. More abstractly, this line represents the separation created by the network between different categories of training examples.

A more complex example using the same principles can be seen below.



Time Series Analysis

A time series can be thought of as chronological set of data points, indexed by time. In more formal terms, a time series is a sequence of vectors,

$$\vec{s}(t), t = 0, 1, ..., \tag{21}$$

where each value of $\vec{s}$ represents some observable variable at the given time t. For example, a meteorological time series might include temperature, humidity, and pressure in $\vec{s}$. From this perspective, we can see that many processes can be modeled as time series, such as stock market closing prices or water consumption in a given community. Neural Networks can be trained to perform a number of tasks with time series, such as classification of behavior, but we will specifically focus on forecasting future values of the series.

While $\vec{s}$ does often represent a continuously defined function of t, it is often more practical within the context of neural networks to instead take $\vec{s}$ as a set of *samples* of the observable variables being tracked, taken at regular intervals. We can then iterate over the series, and have our network predict $\vec{s}$ using some number of previous values, n, from our series. This allows the neural network to take into account the history of the series, and to detect patterns that may affect future outcomes.

Data

We will be training on several types of time series, both real-world and simulated. A sine wave here acts as a simulated time series, for several

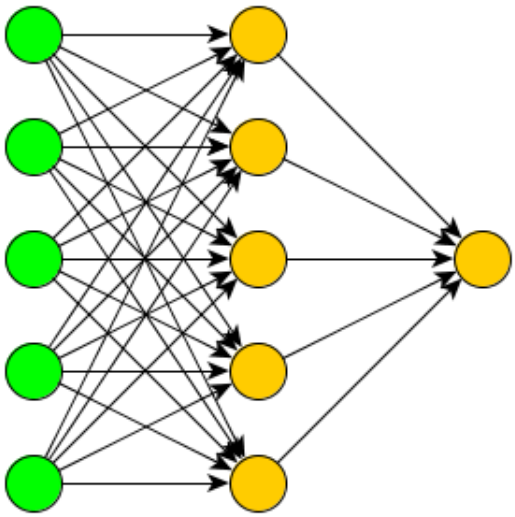
reasons. First, sine waves are cyclic functions, making them similar to many practical patterns we may want to analyze or predict, such as weather patterns. Second, sine waves are very predictable and understood, such that any anomalies in neural network predictions are certainly the fault of the network itself and not due to unusual data. To draw on the notation earlier employed, this would have

$$s(t) = sin(t). \tag{22}$$

Note that in this context we no longer represent s as a vector, since the time series described here only consists of one value at any given point in time. The training set of data was generated with $0 \leq t \leq 10$, sampled at regular .05 intervals. The validation set was generated using the same process, from $10 \leq t \leq 20$. Training examples can be thought of as tuples in the form of $(\vec{x}, l)$, where l is the *label*, or current value of the training example, and the vector $\vec{x}$ holds the n preceding values in the series. Thus, the inputs to the network will be $\vec{x}$, and the expected output, used in training, is l. In our case, training examples consist of 5 samples, meaning all networks used to predict this series will take 5 inputs.

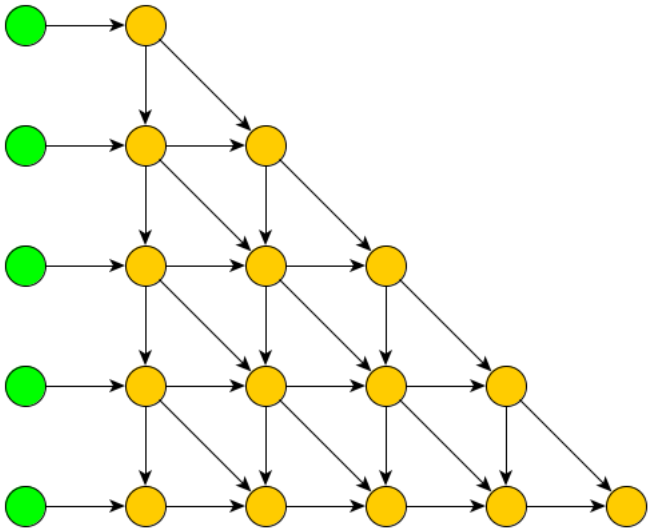
Network Topology

A number of network topologies are used here in the interest of better understanding which techniques and approaches are more accurate or more efficient. The first, simplest approach is to use a fully connected topology with one hidden layer, in this case containing 5 neurons.



A more nuanced approach would be to take into account the chronological nature of the data in a time series. By arranging an alternate topology

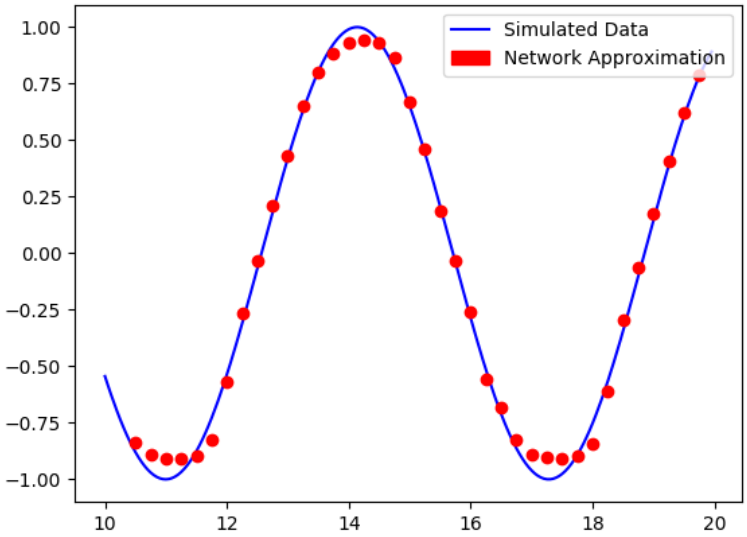
where data flows from older to newer points of data, we can intuitively embody this idea.

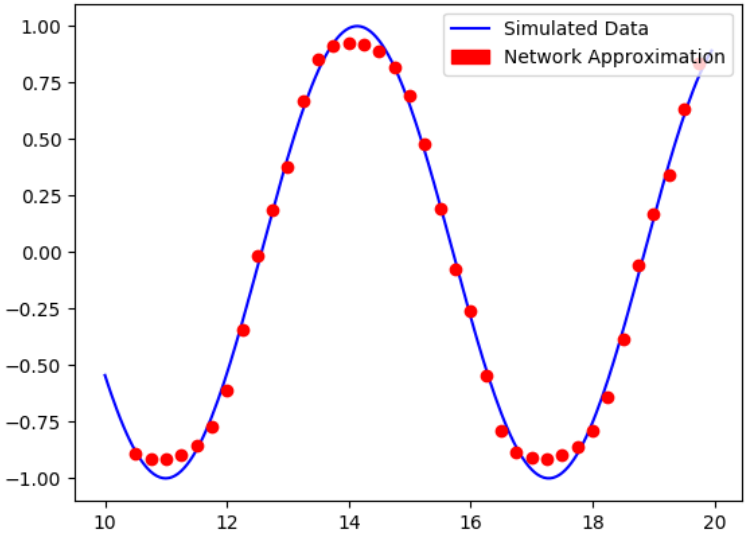


Note also that this layout provides a 'deeper' network, with more neurons between the inputs and outputs, resulting in better ability to represent more nuanced patterns. This model can be extended to any number of inputs by adding additional columns of neurons to the left, each one with one more neuron than the last.

Results

Both networks were fairly proficient in predicting the validation set. Each was trained for 1000 epochs over the training set of 38 examples. The graphs below show the approximations produced from the validation set, along with its actual value (the sine wave). The first is the fully connected network, followed by the triangular network.





One of the most evident issues with these approximations is their failure to reach the peaks and troughs of the function. This is largely due to the choice of activation function, tanh. Since $tanh(x)$ asymptotically approaches 1 as x approaches infinity, and -1 as x approaches negative infinity, the output of the network can never be 1 or -1, and will struggle to approach these values.

References

[1] Arash Ardakani, Carlo Condo, and Warren J. Gross. "Sparsely-Connected Neural Networks: Towards Efficient VLSI Implementation of Deep Neural Networks". In: *CoRR* abs/1611.01427 (2016). arXiv: 1611.01427. URL: http://arxiv.org/abs/1611.01427.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[3] F. Rosenblatt. *Principles of Neurodynamics*. New York: Spartan Books, 1959.