June 2020

Get Ready for AI: Tools to Help You Get Started

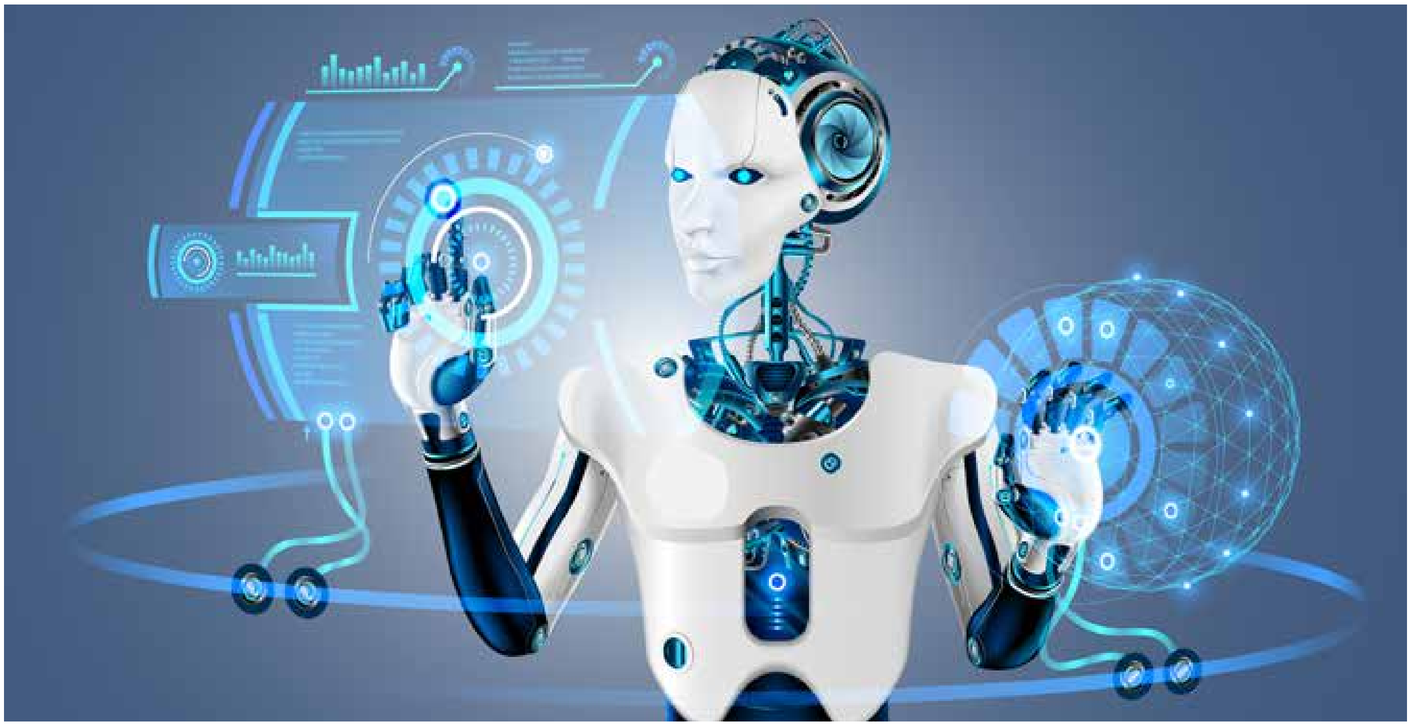
Table of Contents

AI Tools to Help You Get Started

Artificial Intelligence (AI) is a subset of computer science directed towards the development of computers capable of actions usually done by humans. AI as a field of study first emerged during the 1950s, and initial successes included computers proving theorems and playing simple games. Neural networks have subsequently emerged from the concept of intelligent biological computing and have become an important part of AI. This eBook will discuss the basic concepts of AI, including a discussion on Neural Networks, and concluding with a list of AI tools anyone can use to get started.

element14 is a Community of over 700,000 makers, professional engineers, electronics enthusiasts, and everyone in between. Since our beginnings in 2009, we have provided a place to discuss electronics, get help with your designs and projects, show off your skills by building a new prototype, and much more. We also offer online learning courses such as our Essentials series, video tutorials from element14 presents, and electronics competitions with our Design Challenges.

element14 Community Team



Get Ready for AI: Tools to Help You Get Started

CHAPTER 1 Introduction to AI

Artificial intelligence (AI) is a computer science subfield directed towards the development of computers capable of actions usually done by humans.

AI's promise, until now, has far outstripped its deliverability. The latest versions of AI are, however, different. The dissimilarities can be summed up in five drivers: increased computational resources, explosive data growth, focus on particular problems, knowledge engineering, and alternative reasoning models.

The following are the components of AI.

Knowledge: Part of AI which deals with comprehending, designing, and implementing methods of representing information collected from the environment and stored in computers. The agent programs can now make sense of the information and plan future activities. They can also solve problems in areas that generally require human expertise.

Reasoning: A software system that makes decisions from available knowledge using logical methods like presumption and training. The system is termed as Reasoning. Interactive and batch processing are two modes of reasoning. An Interactive system can interface with the user to ask helpful questions and helps the user to guide the reasoning process. Batch systems take into account all the existing information simultaneously and produce the optimum answer without feedback or guidance.

Learning: gaining knowledge or skill by studying, being taught, practicing, or undergoing any action.

Problem Solving: A process in which one first observes and then attempts to reach a preferred solution from a present situation, by adopting a path obstructed by known hurdles or unknown ones. The process encompasses decision making, or selecting the optimal alternative to reach the desired goal.

Perception: Technique of obtaining, comprehending, selecting, and consolidating sensory information. Perception and sensing are intermeshed. Sensory organs support perception. In relation to AI, perception refers to data acquisition by sensors.

Inference: A suitable computational framework for perception and necessary computations. Initially, the neural network goes through training similar to a human brain learning a job. Based on what it has learned, the trained neural network is used to recognize application information like blood disease, images, spoken words, and so on. The system then infers things about new data based on prior learning, and this process is known as inference.

Driving Intelligence with Big Data: AI systems must be demystified to construct a framework for comprehending new systems. AI systems assess, infer, and predict. These systems use data to transact, click links, network connections, and query, among other actions. Big Data enables simple learning systems to filter a signal from the noise. Processing and parallelism then blend to synthesize results.

Assessing Data with AI: Consumer systems assess us. Amazon, for example, collates a meticulous picture to compare us against similar customers and create a predictions source. This is transactional assessment data: what customers touch and what they buy. Amazon's recommendation engines use this information to build profiles and make recommendations. Profile data, however, is only a part of the larger picture. This information includes categories that cluster objects ("cookbooks"), customer categories ("experts"), and information derived from other users. Your spending power and residential address can be funneled in to refine these snapshots of yourself, which also capture the things you interact with. The result is an establishment of characterizations.

CHAPTER 2 What You Need to Know to Get Started

This chapter discusses what a neural network is and how it trains a machine.

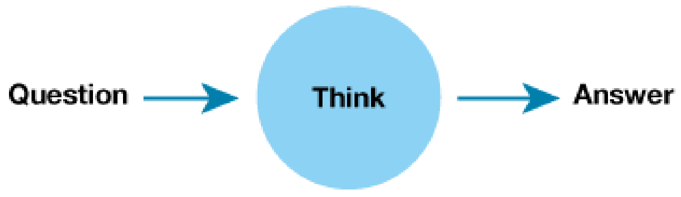
The History and Birth of the Neural Network

AI became a formal field of study during the 1950s. Initial successes included computers proving theorems and playing simple games. Many believed that machines having human-level intelligence would be invented within a decade. This, sadly, did not materialize. A new method was then adopted: copy the biological brain to create an artificial one.

Neural networks subsequently emerged from intelligent biological computing and made inroads in the artificial intelligence discipline. Google's Deepmind, with neural networks in its foundation, is an excellent example. It defeated a human world master at Go, a complex recreational game. Neural networks are employed inside everyday technology, decoding handwritten postcodes on handwritten letters and automatically recognizing car number plates.

A Simple Predicting Machine

Imagine a simple machine that accepts a question, "thinks," and yields an answer. An appropriate human analogy is to receive input through your eyes, use your brain to analyze that scene, and generate a conclusion about the objects present in that scene. Here's a pictorial representation in diagram to the right.

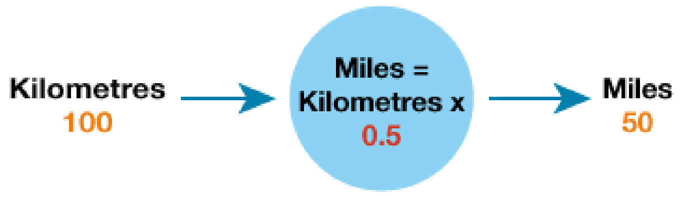


Computers do not think. A machine accepts some input, does a little calculation, and submits an output.

Imagine a machine which converts kilometers to miles. It is known that the relationship between these two is linear. It implies that if we double the number of miles, the kilometers distance also gets increased. Such a linear relationship between kilometers and miles provides a clue about the needed calculation in the form "miles = kilometers x c," where c is constant. This constant c, at present, continues to be unknown for the purposes of this example.

To work out this missing constant c, you can pluck a random value, such as c = 0.5, and see the result.

Here miles = kilometers x c, where kilometers is 100 and c is 0.5.



The result is 50 miles. We know c = 0.5 is incorrect, as the correct answer is 62.137. The difference between right and wrong is 12.137. This is the error, the difference between actual and our calculated answer.

Thus, error = truth - calculated

= 62.137 - 50

= 12.137

Now, if we try again with c=0.6, the error is a much reduced 2.137. Moving to 0.7 will show the error to be -7.863. The minus sign states that we have overshot. Values between 0.6 and 0.7 must be checked. This clearly illustrates the impossibility of solving a problem in a single step.

Classifying

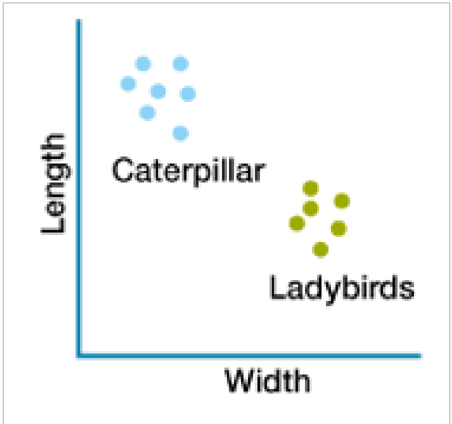
Classifying is the same as predicting.

The plot 1 graph shows the measured dimensions of garden bugs.

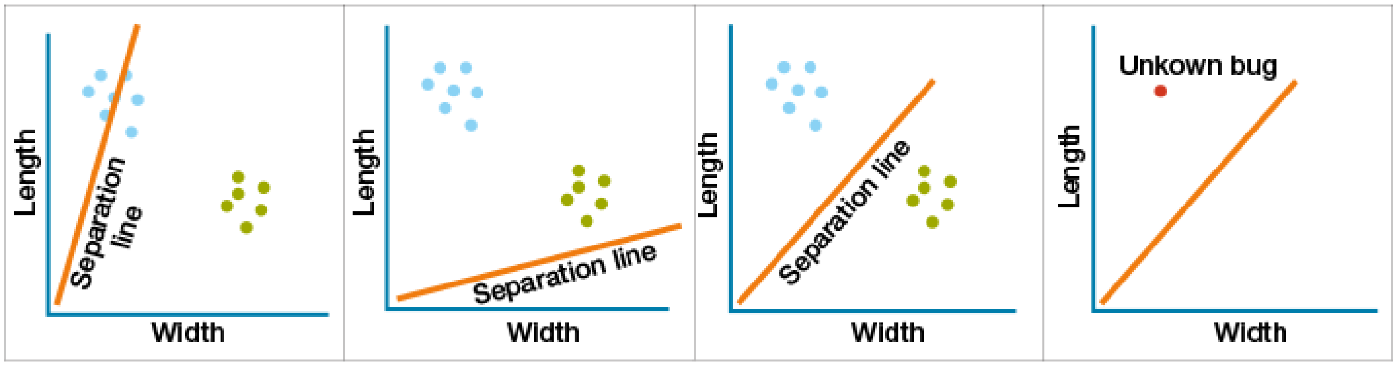
Two groups include thin and long caterpillars and wide and short ladybirds.

Remember the predictor which attempted to calculate the correct number of miles if provided kilometers? That predictor was armed with an adjustable linear function in its heart. It is worth noting that linear functions provide straight lines when you plot output against an input. This adjustable parameter c altered the straight line slope.

The splitting line in plot 2 and plot 3 cannot be a good classifier, as one is on the caterpillar, and the other is under the ladybird. Plot 4 displays the good classifier separating the two. It can be utilized to identify the new bug, as displayed in plot 5.



Plot1: Garden Bug Dimension



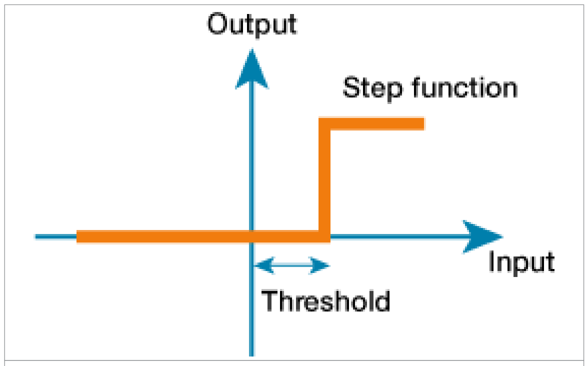
Plot 2 Plot 3 Plot 4 Plot 5

Neurons

Neurons–in all their forms—transmit electrical signals along their length, from dendrites along the axons to terminals. These signals are transmitted between neurons. This is how a human body senses light, touch pressure, sound, and so on. Specialized sensory neurons transmit signals along the nervous system to the brain, which itself is mostly neurons.

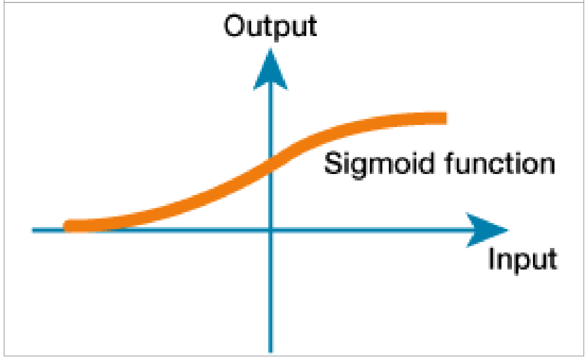
A neuron works by taking an electric input and popping out an electrical signal. This looks identical to the older classifying or predicting machines, which accepted an input, executed some processes, and gave an output.

Observations imply that neurons suppress input until it has grown so big that it activates an output. This can be thought of as a threshold that must be touched before the production of any output.



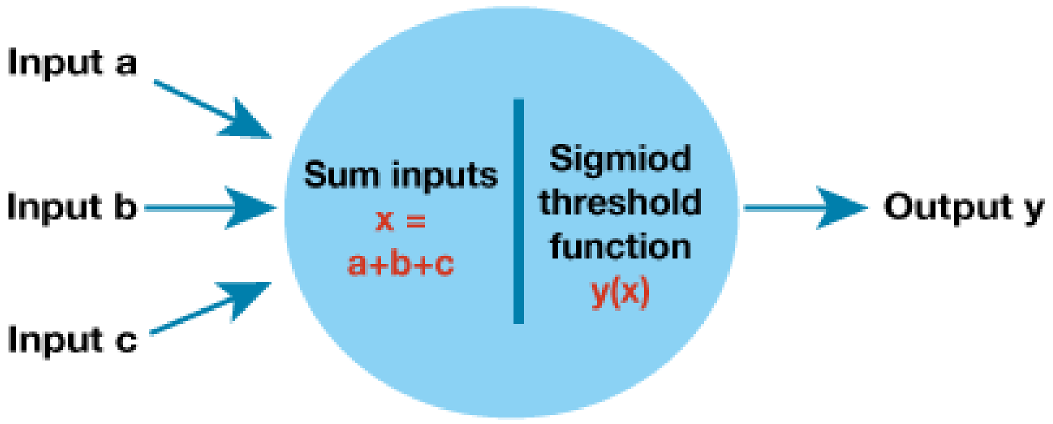
The diagram to the left features a good example of an undemanding step function.

It is observed that output is zero for low input values. The output spikes when threshold input is touched. An artificial neuron behaving identically would replicate a biological neuron. Scientists use the phrase “neurons firing” when the input reaches the threshold.



The smooth S-shaped sigmoid function will be used for making our neural network. Artificial intelligence researchers use similar functions. The sigmoid is a simple and common function, and is termed a logistic function.

The following diagram shows this idea of merging inputs and subsequent application of this threshold to the amalgamated sum:



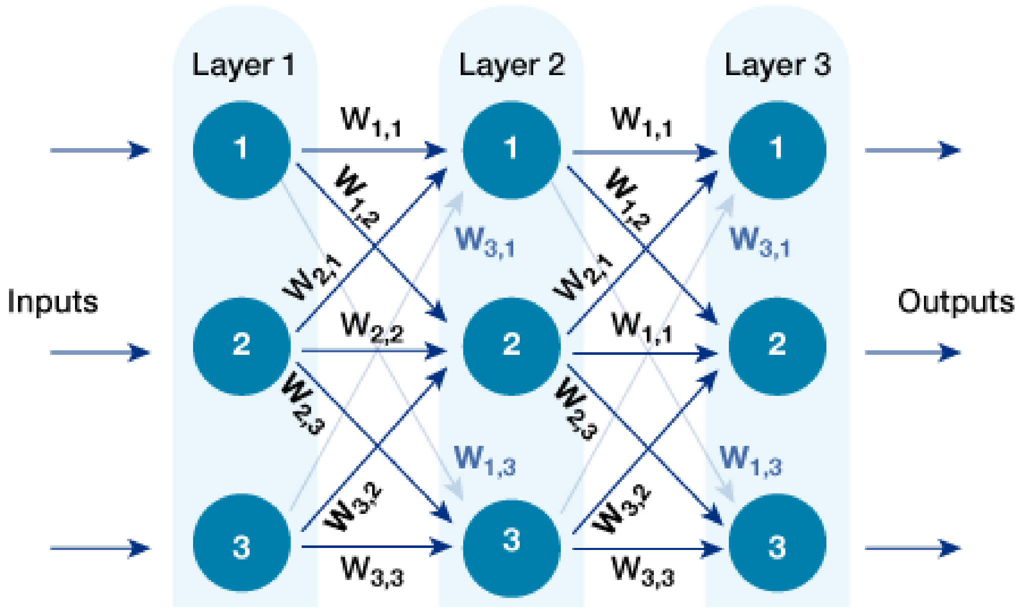
If the merged signal is insufficiently large, then the sigmoid threshold function causes output signal suppression. A large sum x causes the sigmoid to fire that neuron. It is to be noted that even if one of several inputs is large and the balance small, it is enough to fire the neuron.

Dendrites collate the electrical signals and combine to make a powerful electrical signal. If this signal is sufficiently strong to sweep the threshold, dendrites fire a signal through the axon to the terminals to flow on to the succeeding neuron’s dendrites.

Replicating these natural phenomena in an artificial model requires neurons in layers, with each neuron interconnected to all others in the subsequent and preceding layer. The following diagram explains this idea:

You can see three layers, each with three nodes or artificial neurons. Each node is connected to all other nodes in the preceding and succeeding layers.

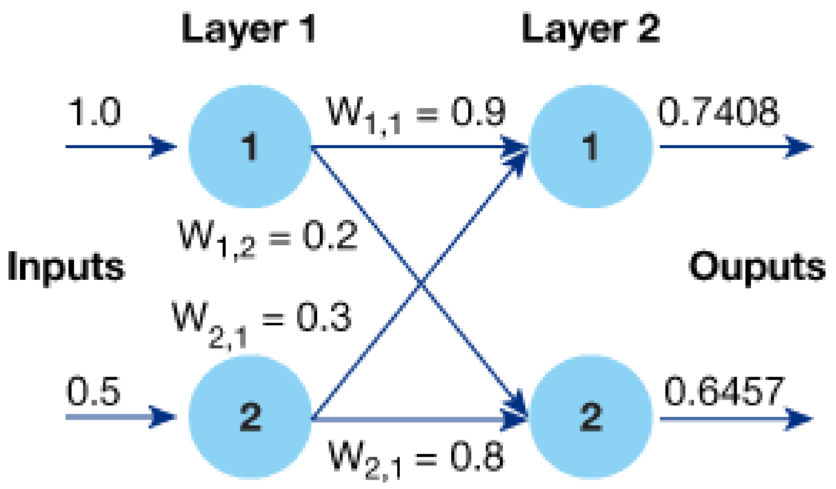
It is logical to adjust the robustness of the connections joining the nodes. The summation of the inputs can be adjusted inside the node, or the sigmoid threshold function shape is altered. The latter, however, is complicated compared to simply changing the strength of the node links.



The figure above shows a weight related to each connection. A high weight will result in boosting the signal, while a low weight will quiet it.

Following a Signal Through a Neural Network

The following illustration explains a smaller, two layer neural network, with each layer having two neurons:



Neural Network Signal Flow

Let's assume the two inputs are 1.0 and 0.5. These inputs are entered into this smaller neural network.

A few random weights are assumed:

- w 1,1 = 0.9
- w 1,2 = 0.2
- w 2,1 = 0.3
- w 2,2 = 0.8

It is now prudent to calculate the output of both inputs. The output x is given as

X = (output from first node * link weight) + (output from second node* link weight)

X1= (1.0 * 0.9) + (0.5 * 0.3)=1.05

X2=(1.0 * 0.2) + (0.5 * 0.8)=0.6

Applying the sigmoid function we get

Y(X1)=1/(1+0.34993)=1/(1.34993)=0.7408

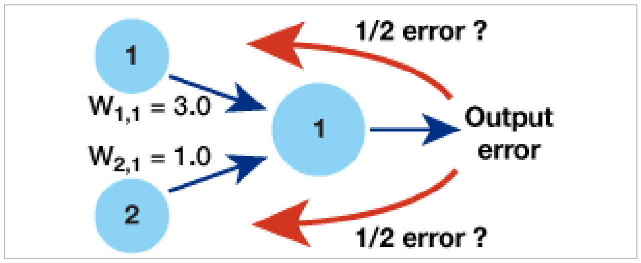
Y(X2)= 1/(1 + 0.5488) = 1/(1.5488) = 0.6457

These simple calculations will be too complicated for a network comprised of 5 layers and 100 nodes in every layer. Matrices can solve this puzzle. The above calculation can be written in matrix form as:

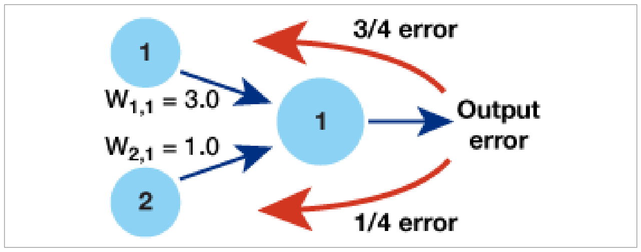
$$\begin{pmatrix} W_{1,1} & W_{2,1} \\ W_{1,2} & W_{2,2} \end{pmatrix} \begin{pmatrix} input1 \\ input2 \end{pmatrix} = \begin{pmatrix} (input1 \times W_{1,1}) + (input2 \times W_{2,1}) \\ (input1 \times W_{1,2}) + (input2 \times W_{2,2}) \end{pmatrix}$$

Learning Weights from More Than One Node

A simple linear classifier can be refined by adjusting the node's linear function. The error (the difference between the answer generated by the node and what we know the answer should be) guides that refinement. This is not difficult, as the relationship between error and required slope adjustment can be easily calculated.



The problem of updating link weights comes to the fore when several nodes contribute to the output and also its error. One idea is to cut this error equally amongst all the contributing nodes.



Another approach is to chop the error unequally. More error is given to those contributing connections that have additional link weights. The reason? Their error contribution is more significant. The diagram to the left demonstrates this idea.

Here two nodes contribute a signal to output node with link weights 3.0 and 1.0. A proportionate splitting of the error to these weights reveals ¾ of output error to be used for more significant weight update, and ¼ of error for the residual smaller weight.

This idea can be extended to other nodes. If 100 nodes are linked to an output node, the error is split across 100 connections to the output node in proportion to each link's contribution to that error, as indicated by link weight size.

Weights are thus used in two ways. The weights at first communicate signals from input to output layers within a neural network. The weights are then used to reverse transmit the error from the output into the network. This method is termed backpropagation.

Neural Network with Python

We can write our neural network with Python and incrementally build up a Python program.

The Skeleton Code

We must sketch out how a neural network class should appear. There must be a minimum of three functions:

- Initialization to set number of input, hidden, and output nodes
- Train refine weights after a training set example is given for learning purposes
- Generate an answer from output nodes after input is given in a query

The shape of the code would be like:

```
# neural network class definition
class neuralNetwork :
# initialize the neural network
def _init__() :
pass
# train the neural network
def train() :
pass
# query the neural network
def query() :
pass
```

Initializing the Network

The number of input, hidden, and output layer nodes must be set to define the neural network's shape and size. These are not permanent, and can be set when parameters can be used to create a new neural network object. This procedure retains the option to create new different-sized neural networks easily.

The developed neural network code must accept a maximum number of useful open options, and simultaneously keep assumptions to a bare minimum. If this is done, the code can be effortlessly used for multiple needs. The class, which can create a small neural network, can also create a huge one by merely passing the necessary size as parameters. The learning rate—another useful parameter—is set during the creation of a brand new neural network.

Weights are the Heart of the Network

The next step constitutes the creation of the nodes and links network. Link weights are a vital component of this network, and are used to calculate the specific signal being fed forward, the error as it is propagated backward, and also the link weights themselves, which are refined to enhance the network.

Since weights can be concisely conveyed as a matrix, it is possible to create:

- A matrix for weights for links between input and hidden layers, W input_hidden, of size(hidden_nodes by input_nodes).
- A second matrix for links between hidden and output layers, W hidden_output, of size (output_nodes by hidden_nodes).
- The older convention is to investigate why the first matrix is of size (hidden_nodes by

input_nodes) and the other way around (input_nodes by hidden_node).

The link weights' initial values must be small. The NumPy function produces an array of values selected randomly between 0 and 1, where size is (rows by columns).

Querying the Network

The query () function draws input to a neural network and returns the network's output. To do this, the input signals must pass from the nodes' input layer via the hidden layer and exit from the final output layer. It is to be noted that link weights are used to moderate signals as they are fed into any output or hidden node. The sigmoid activation function is used to output the signal arising from those nodes.

A vast population of nodes implies the painful task of writing Python code for each node, conducting weight moderation, summing the signals, and applying an activation function. More nodes mean more code. There is, however, no need, as all these instructions can be written in a concise and straightforward matrix form.

The following content expands on how the matrix of weights for that link between the simple Python piece combines all inputs with the right link weights to generate the combined moderated signals matrix into each hidden layer node. Rewriting is unnecessary if we select a different number of nodes for hidden or input layers to use the next time. The sigmoid squashing function is added to each of these emerging signals:

O hidden = sigmoid(X hidden) to get the signals from the hidden node.

The sigmoid function is defined in the Python library. The scipy Python library contains a special function set, and the sigmoid function is termed expit().

Since we want to tweak or make a radical change to the activation function, it is imperative to define it once inside the neural network object during its first initialization. We can subsequently refer to it multiple times, as in the query () function. This arrangement implies a single definition change, and it is unnecessary to locate and then edit the code anywhere where an activation function is used.

The activation function in use within the neural network's initialization section is defined by:

```
# activation function is the sigmoid function

self.activation_function = lambda x: scipy.special.expit(x)
```

Training the Network

The training task needs more involvement and has two parts:

The first part involves finding the output for any given training example; the second involves taking the calculated output, comparing it with the desired output, and using this difference to guide network weights.rk updation.

The weights must be improved during training based on the error between the calculated output and the target output. This can be done through manageable steps.

The first action is to calculate the error, which is the difference between the wanted target output offered by the specific training example and the actual calculated output. This is the difference between the matrices targetsfinal_outputs) done by combining through every element.

The Complete Neural Network Code

```
# neural network class definition
class neuralNetwork :
    # initialise the neural network
    def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate) :
        # set number of nodes in each input, hidden, output layer
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes
        # link weight matrices, wih and who
        # weights inside the arrays are w_i_j, where link is from node i to node j in the next layer
        # w11 w21
        # w12 w22 etc
        self.wih = numpy.random.normal(0.0, pow(self.hnodes, 0.5), (self.hnodes, self.inodes))
        self.who = numpy.random.normal(0.0, pow(self.onodes, 0.5), (self.onodes, self.hnodes))
        # learning rate
        self.lr = learningrate
        # activation function is the sigmoid function
        self.activation_function = lambda x: scipy.special.expit(x)
        pass
    # train the neural network
    def train(self, inputs_list, targets_list) :
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T
        targets = numpy.array(targets_list, ndmin=2).T
        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)
        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)
        # output layer error is the (target-actual)
        output_errors = targets-final_outputs
        # hidden layer error is the output_errors, split by weights, recombined at hidden nodes
        hidden_errors = numpy.dot(self.who.T, output_errors)
        # update the weights for the links between the hidden and output layers
        self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.
transpose(hidden_outputs))
        # update the weights for the links between the input and hidden layers
        self.wih += self.lr * numpy.dot((hidden_errors *hidden_outputs * (1.0 - hidden_outputs)), numpy.
transpose(inputs))
        pass
        # query the neural network
    def query(self, inputs_list) :
        # convert inputs list to 2d array
        inputs = numpy.array(inputs_list, ndmin=2).T
        # calculate signals into hidden layer
        hidden_inputs = numpy.dot(self.wih, inputs)
        # calculate the signals emerging from hidden layer
        hidden_outputs = self.activation_function(hidden_inputs)
        # calculate signals into final output layer
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # calculate the signals emerging from final output layer
        final_outputs = self.activation_function(final_inputs)
        return final_outputs
```

CHAPTER 3 Tools to Help You Get Started

Machine learning (ML) tools can be used to help get your ML project started.

Platform Versus Library

Machine learning tools can be separated into Libraries and Platforms. A platform offers everything to run a project, whereas a library solely provides discrete capabilities or the parts needed for project completion.

Examples of ML platforms:

- Amazon Web Services
- Azure machine learning tools
- IBM's Watson platform

ML Libraries:

The following table shows popular libraries used in ML:

Purpose	Library
Scientific Computation	Numpy
Tabular Data	Pandas
Data Modelling & Preprocessing	Scikit Learn
Deep Learning	Tensorflow, Pytorch, Caffe

While ML platforms focus primarily on business or end product implementation, the library is recommended for enthusiasts who want to write code from scratch and hone their subject skills. To start, the user must possess a working knowledge of Python and ML libraries. The user must have the skill to use these libraries installed within the system.

Devices for Inferencing

ML libraries can be easily installed on several Single Board Computers (SBCs) like the BeagleBone AI, Avnet Ultra96-V2 dev board, or the Raspberry Pi. These SBCs are ideal for various inferencing tasks, with the model being deployed on such devices after creation and training. It is recommended that you should not train any model on SBCs, since it involves complex matrix computation. There is an additional complication, too: a powerful GPU is necessary to train a complex model.

The Raspberry Pi has considerable community support, and ML libraries are easily accessed. Google's Tensorflow library officially supports the Raspberry Pi.

The article **Machine Learning: Using Tensorflow on the Raspberry Pi**, gives an example of how to use this ML library with the Pi. See article here.

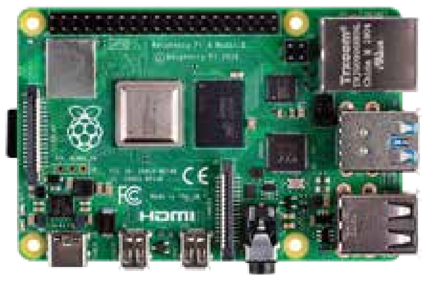
The BeagleBone AI comes with a preinstalled Caffe-Jacinto library based on Caffe, which helps to speed up the task as new code does not need to be written from scratch. The BeagleBone AI also comes with an embedded-vision-engine (EVE) core, enabling efficient processing of videos and images.

The inferencing task occasionally requires better computation capabilities along with superior graphic processing. The Avnet Ultra96-V2 board is a good choice, and comes with an Arm processor and an FPGA for better processing. The board's neural processing unit (NPU) contributes to efficient neural network computation.

The Brainium SmartEdge Agile device can also be used when you want to deploy an ML model without getting deep into the weeds. The Branium portal is an ML Platform permitting users to create an ML model without writing a single slice of code. The result can be directly used on the SmartEdge Agile device.

BeagleBone® AI

The BeagleBone® AI is a high-end Single Board Computer aimed at developers interested in implementing machine-learning and computer vision with simplicity. BeagleBone® AI includes a dual-core ARM Cortex-A15 running at 1.5 GHz, 16GB on-board eMMC flash, a SuperSpeed USB Type-C interface, Gigabit Ethernet and dual band wireless connectivity.

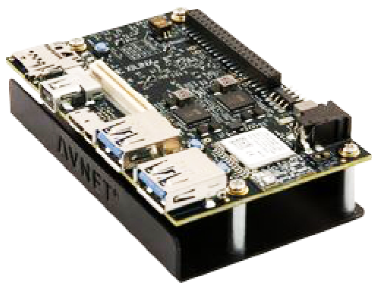


Raspberry Pi 4 Model B

Raspberry Pi 4 Model B offers ground-breaking increases in processor speed, multimedia performance, memory, and connectivity. This product's key features include a high-performance 64-bit quad-core processor, dual-display support at resolutions up to 4K via a pair of micro-HDMI ports, hardware video decode at up to 4Kp60, 2GB of RAM, dual-band 2.4/5.0GHz wireless LAN, Bluetooth 5.0, Gigabit Ethernet, USB 3.0.   Also available in 4GB and 8GB.

SMARTEDGE AGILE

The SMARTEDGE AGILE meta-sensor, together with the Brainium IoT platfrom, form part of an end to end IoT solution that delivers AI and Security at the Edge. Brainium accelerates delivery of IoT solutions with higher performance, security and stability, while significantly reducing costs.



Ultra96-V2

The Avnet Ultra96-V2 is an Arm-based, Xilinx Zynq UltraScale+ ™ MPSoC development board based on the Linaro 96Boards Consumer Edition (CE) specifications, providing, developers with a unique and powerful environment to simplify machine learning. See the Industrial version.

SmartEdge Industrial IoT Gateway

Avnet's SmartEdge Industrial IoT Gateway is based on the Raspberry Pi's Broadcom BCM2837 SoC, a 64-Bit quad-core ARM processor and features on-board 8GB eMMC, RS485/RS232, CAN & Modbus interfaces and a -20°C to 70°C temperature range.

See more available AI products in the Americas, Europe, or Asia-Pacific.

element14
AN AVNET COMMUNITY

300 S. Riverside Plaza, Suite 2200
Chicago, IL 60606

www.element14.com/community

Facebook.com/e14Community
Twitter.com/e14Community