NEURAL.club

Artificial intelligence (AI), deep learning, and neural networks represent incredibly exciting and powerful machine learning-based techniques used to solve many real-world problems.

While human-like deductive reasoning, inference, and decision-making by a computer is still a long time away, there have been remarkable gains in the application of AI techniques and associated algorithms.

➔ Machine learning	➔ Blockchain microfinances	➔ API integration
➔ Artificial intelligence	➔ Distribution Decentralized	➔ Science mining
➔ Data science	➔ Nvidia[tm] CUDA[tm] & cuDNN	➔ Data mining
➔ Predictive analytics	➔ Google[tm] Tensorflow[tm]	➔ GUI & SDK

OPEN SOURCE COLLECTIVE ARTIFICIAL INTELLIGENCE BLOCKCHAIN BASED

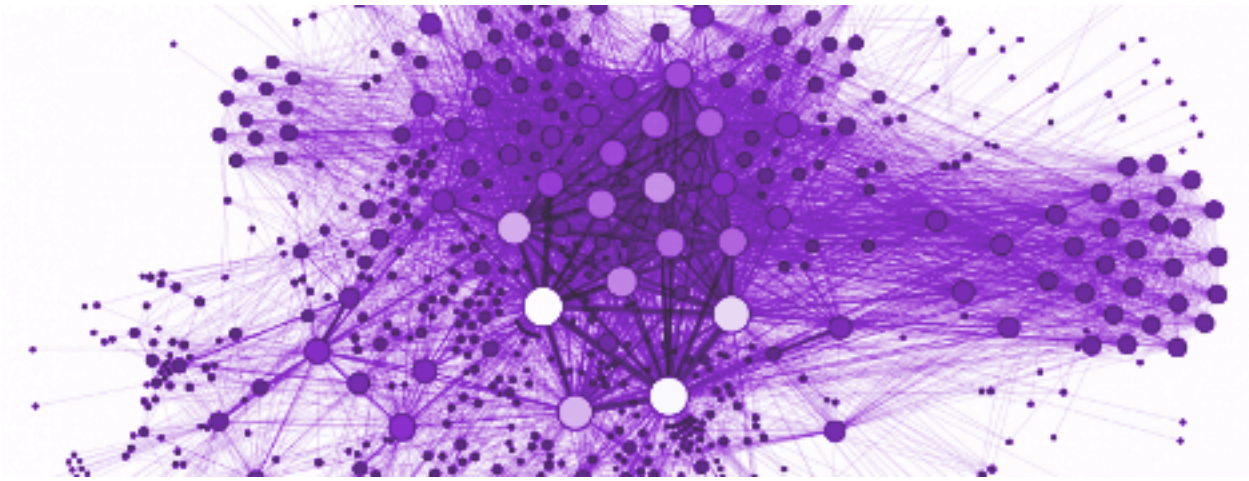
Content

The future is here and now

Machine learning is a very hot topic for many key reasons, and because it provides the ability to automatically obtain deep insights, recognize unknown patterns, and create high performing predictive models from data, all without requiring explicit programming instructions.

Despite the popularity of the subject, machine learning's true purpose and details are not well understood, except by very technical folks and/or data scientists.

The concepts of our project is blockchain database, it collect all final knowledge based on mathematics, statistics, probability theory, physics, signal processing, machine learning, computer science, psychology, linguistics, and neuroscience.



This paper is intended to be a comprehensive, in-depth, and non-technical guide to machine learning, and should be useful to everyone from business executives to machine learning practitioners. It covers virtually all aspects of machine learning (and many related fields) at a high level, and should serve as a sufficient introduction or reference to the terminology, concepts, tools, considerations, and techniques of the field.

This high level understanding is critical if ever involved in a decision-making process surrounding the usage of machine learning, how it can help achieve business and project goals, which machine learning techniques to use, potential pitfalls, and how to interpret the results.

Machine Learning (ML) has its origins in the field of Artificial Intelligence, which started out decades ago with the lofty goals of creating a computer that could do any work a human can do.

Machine Learning Algorithms

We've now covered the machine learning problem types and desired outputs. Now we will give a high level overview of relevant machine learning algorithms.

Supervised Regression

- Simple and multiple linear regression
- Decision tree or forest regression
- Artificial Neural networks
- Ordinal regression
- Poisson regression
- Nearest neighbor methods (e.g., k-NN or k-Nearest Neighbors)

Supervised Two-class & Multi-class Classification

- Logistic regression and multinomial regression
- Artificial Neural networks
- Decision tree, forest, and jungles
- SVM (support vector machine)
- Perceptron methods
- Bayesian classifiers (e.g., Naive Bayes)
- Nearest neighbor methods (e.g., k-NN or k-Nearest Neighbors)
- One versus all multiclass

Unsupervised

- K-means clustering
- Hierarchical clustering

Anomaly Detection

- Support vector machine (one class)
- PCA (Principle component analysis)

Note that a technique that's often used to improve model performance is to combine the results of multiple models. This approach leverages what's known as ensemble methods, and random forests are a great example (discussed later).

If nothing else, it's a good idea to at least familiarize yourself with the names of these popular algorithms, and have a basic idea as to the type of machine learning problem and output that they may be well suited for.

Summary

Machine learning, predictive analytics, and other related topics are very exciting and powerful fields.

While these topics can be very technical, many of the concepts involved are relatively simple to understand at a high level. In many cases, a simple understanding is all that's required to have discussions based on machine learning problems, projects, techniques, and so on.

Chapter two of this series will provide an introduction to model performance, cover the machine learning process, and discuss model selection and associated tradeoffs in detail.

NEURAL

Algorithms & methods used by Data Scientists.

Comparing with 2011 Poll Algorithms for data analysis / data mining we note that the top methods are still Regression, Clustering, Decision Trees/Rules, and Visualization. The biggest relative increases, measured by (pct2016 /pct2011 - 1) are for

- **Boosting**, up 40% to 32.8% share in 2016 from 23.5% share in 2011
- **Text Mining**, up 30% to 35.9% from 27.7%
- **Visualization**, up 27% to 48.7% from 38.3%
- **Time series/Sequence analysis**, up 25% to 37.0% from 29.6%
- **Anomaly/Deviation detection**, up 19% to 19.5% from 16.4%
- **Ensemble methods**, up 19% to 33.6% from 28.3%
- **SVM**, up 18% to 33.6% from 28.6%
- **Regression**, up 16% to 67.1% from 57.9%

Most popular among new options added in 2016 are

- K-nearest neighbors, 46% share
- PCA, 43%
- Random Forests, 38%
- Optimization, 24%
- Neural networks - Deep Learning, 19%
- Singular Value Decomposition, 16%

The biggest declines are for

- Association rules, down 47% to 15.3% from 28.6%
- Uplift modeling, down 36% to 3.1% from 4.8% (that is a surprise, given strong results published)
- Factor Analysis, down 24% to 14.2% from 18.6%
- Survival Analysis, down 15% to 7.9% from 9.3%

The following table shows usage of different algorithms types: Supervised, Unsupervised, Meta, and other by Employment type. We excluded NA (4.5%) and Other (3%) employment types.

Table 1: Algorithm usage by Employment Type

Employment Type	% Voters	Avg Num Algorithms Used	% Used Super-vised	% Used Unsuper-vised	% Used Meta	% Used Other Methods
Industry	59%	8.4	94%	81%	55%	83%
Government/Non-profit	4.1%	9.5	91%	89%	49%	89%
Student	16%	8.1	94%	76%	47%	77%
Academia	12%	7.2	95%	81%	44%	77%
All		8.3	94%	82%	48%	81%

We note that almost **everyone uses supervised learning algorithms**.
Government and Industry Data Scientists used **more different types of algorithms** than students or academic researchers,
and **Industry Data Scientists were more likely to use Meta-algorithms**.

Next, we analyzed the usage of top 10 algorithms + Deep Learning by employment type.

Table 2: Top 10 Algorithms + Deep Learning usage by Employment Type

Algorithm	Industry	Government/Non-profit	Academia	Student	All
Regression	71%	63%	51%	64%	67%
Clustering	58%	63%	51%	58%	57%
Decision	59%	63%	38%	57%	55%
Visualization	55%	71%	28%	47%	49%
K-NN	46%	54%	48%	47%	46%
PCA	43%	57%	48%	40%	43%
Statistics	47%	49%	37%	36%	43%
Random Forests	40%	40%	29%	36%	38%
Time series	42%	54%	26%	24%	37%
Text Mining	36%	40%	33%	38%	36%
Deep Learning	18%	9%	24%	19%	19%

To make the differences easier to see, we compute the algorithm bias for a particular employment type relative to average algorithm usage as Bias(Alg,Type)=Usage(Alg,Type)/Usage(Alg,All) - 1.

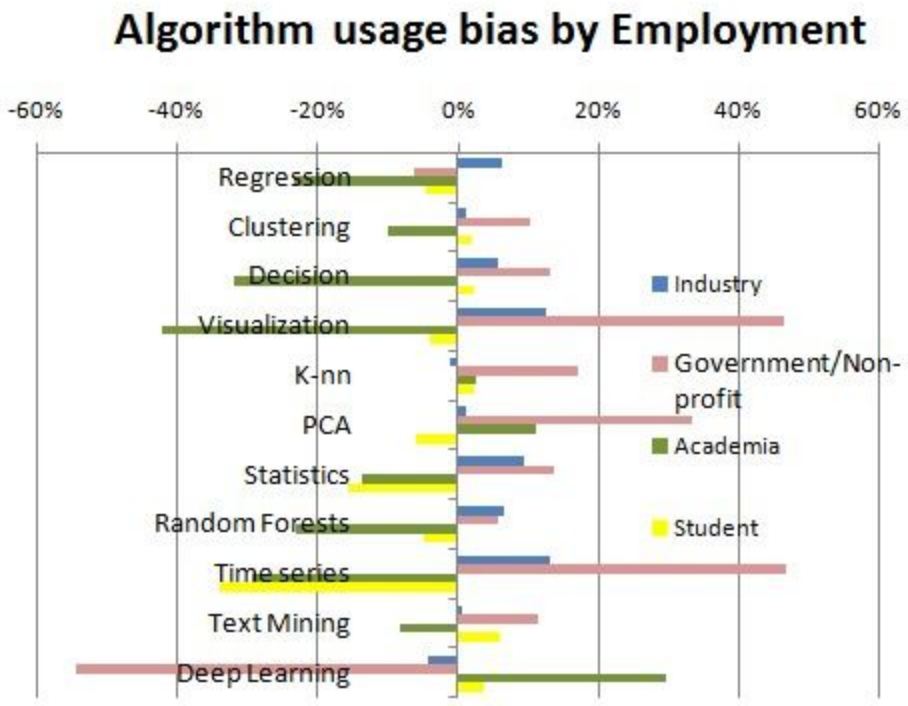
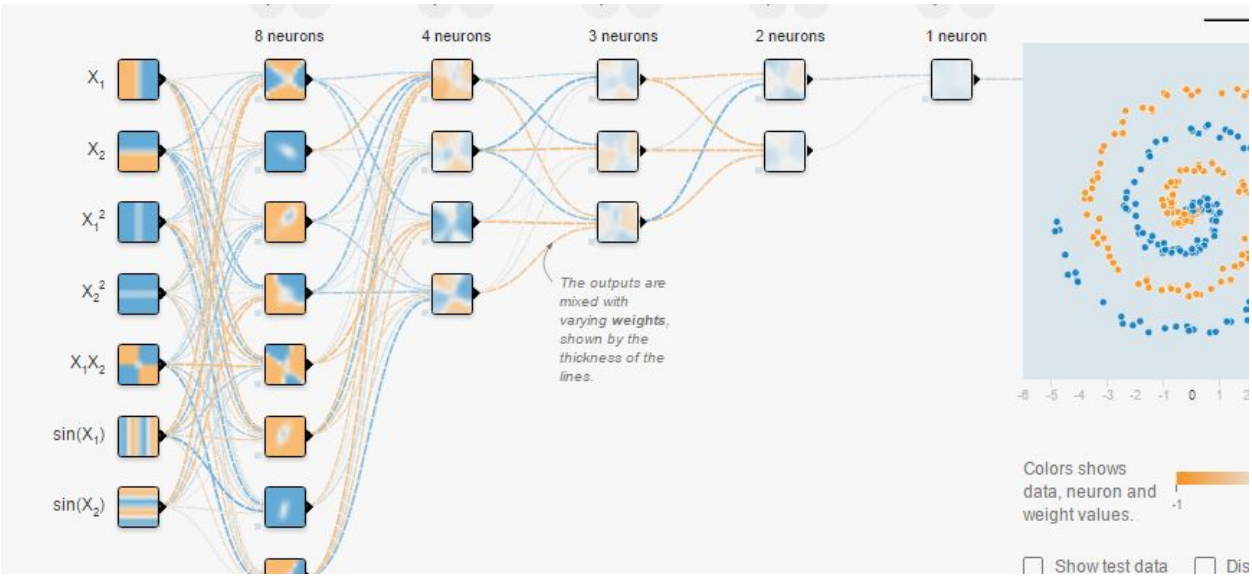


Fig. 1: Algorithm usage bias by Employment.

We note that Industry Data Scientists are more likely to use Regression, Visualization, Statistics, Random Forests, and Time Series. Government/non-profit are more likely to use Visualization, PCA, and Time Series. Academic researchers are more likely to use PCA and Deep Learning. Students generally use fewer algorithms, but do more text mining and Deep Learning.



Parallelism in Machine Learning: GPUs, CUDA, and Practical Applications

The lack of parallel processing in machine learning tasks inhibits economy of performance, yet it may very well be worth the trouble. Read on for an introductory overview to GPU-based parallelism, the CUDA framework, and some thoughts on practical implementation.

Traditionally (whatever that means in this context), machine learning has been executed in single processor environments, where algorithmic bottlenecks can lead to substantial delays in model processing, from training, to classification, to distance and error calculations, and beyond. Beyond recent technology-harnessing in neural networking training, much of machine learning - including both off-the-shelf libraries like scikit-learn and DIY algorithm implementation - has been approached without the use of parallel processing.

The lack of parallel processing, in this context referring to parallel execution on a shared-memory architecture, inhibits the potential exploitation of large numbers of concurrently-executing threads performing independent tasks in order to achieve economy of performance. The dearth of parallelism is attributable to all sorts of reasons, not the least of which being that parallel programming is **hard**. It really is.

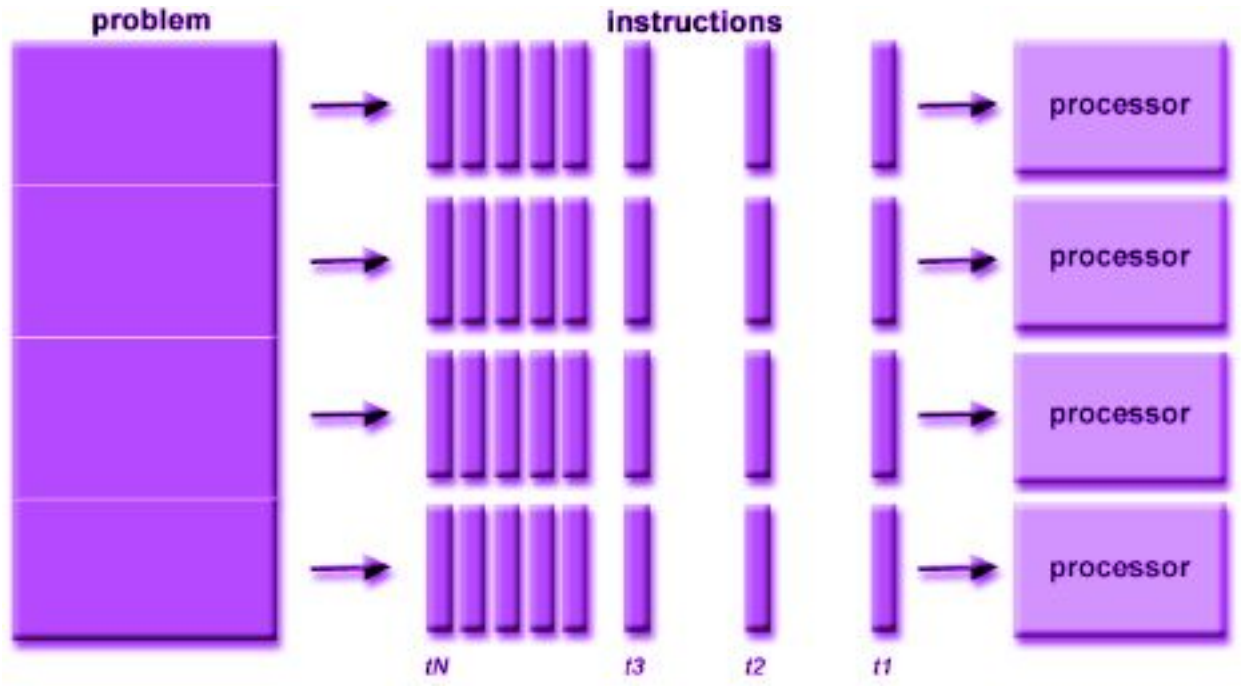


Fig. 1: Parallel Problem Overview.

Also, parallel processing is not magic, and cannot "just be used" in every situation; there are both practical and theoretical algorithmic design issues that must be considered when even thinking about incorporating parallel processing into a project. However, with Big Data encompassing such large amounts of data, sets of which are increasingly being relied upon for routine machine learning tasks, the trouble associated with parallelism may very well be worth it in a given situation due solely to the potential of dramatic time-savings related to algorithm execution.

General Purpose Computing on Graphics Processing Units

A contemporary favorite for parallelism **in appropriate situations**, and focus of this article, is utilizing general purpose computing on graphics processing units (GPGPU), a strategy exploiting the numerous processing cores found on high-end modern graphics processing units (GPUs) for the simultaneous execution of computationally expensive tasks. While not all machine learning tasks, or any other collection of software tasks for that matter, can benefit from GPGPU, there are undoubtedly numerous computationally expensive and time-monopolizing tasks to which GPGPU **could** be an asset. Modifying algorithms to allow certain of their tasks to take advantage of GPU parallelization can demonstrate noteworthy gains in both task performance and completion speed.

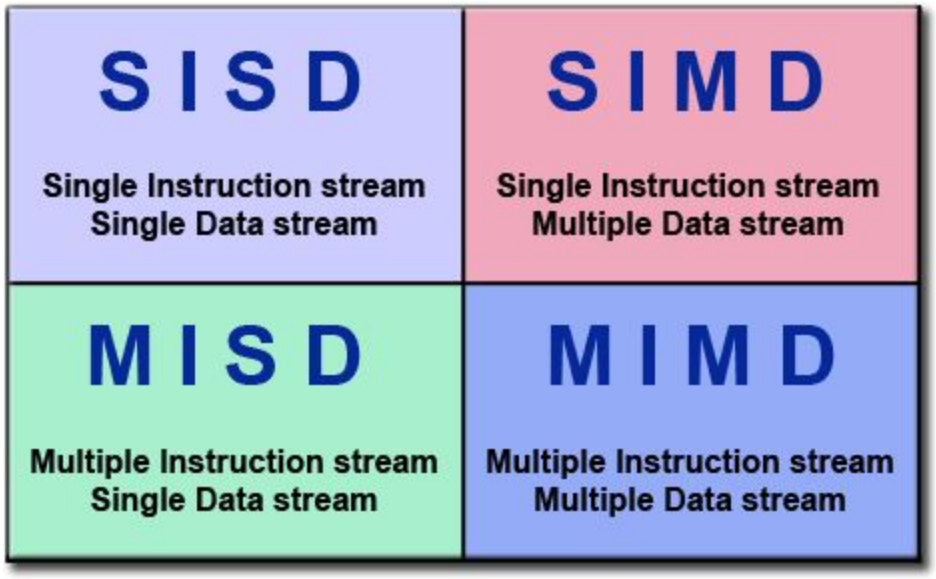


Fig. 2: Flynn's Taxonomy.

The GPGPU paradigm fits into Flynn's taxonomy as single program, multiple data (SPMD) architecture, which differs from from the traditional multi core CPU computing paradigm.

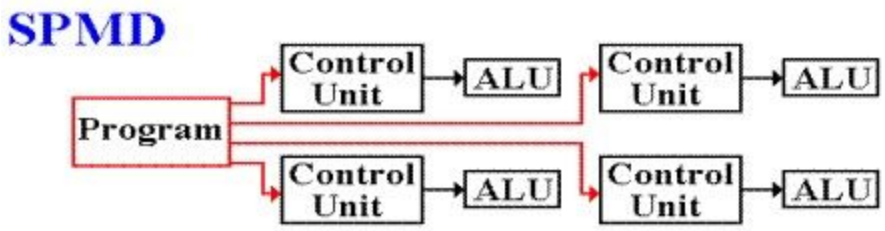


Fig. 3: Single Program Multiple Data (SPMD) subdivision of MIMD.

It should be noted that, while these modifications would undoubtedly benefit the processing of the very large datasets which are the very definition of Big Data, their implementation could have a positive effect on much smaller sets of data as well. A number of particular machine learning tasks can be computationally expensive and time-consuming regardless of data size. Parallelizing those which are not necessarily required to be executed in serial could **potentially** lead to gains for small datasets as well.

Machine learning algorithms could also see performance gains by parallelizing common tasks which may be shared among numerous algorithms, such as performing matrix multiplication, which is used by several classification, regression, and clustering techniques, including, of particular interest, linear regression.

An interesting sidenote relates to the theoretical expected speedup in task execution latency. Amdahl's Law states that the theoretical speedup of an entire task's execution increases with the incremental improvement of each system resource. However, regardless of the collective improvement's magnitude, theoretical speedup is limited by the constituent task which cannot benefit from parallel improvements, or improves the least. The chain is only as strong (fast) as its weakest (slowest) link.

For an in-depth introductory treatment of generalized parallel computing, read this.

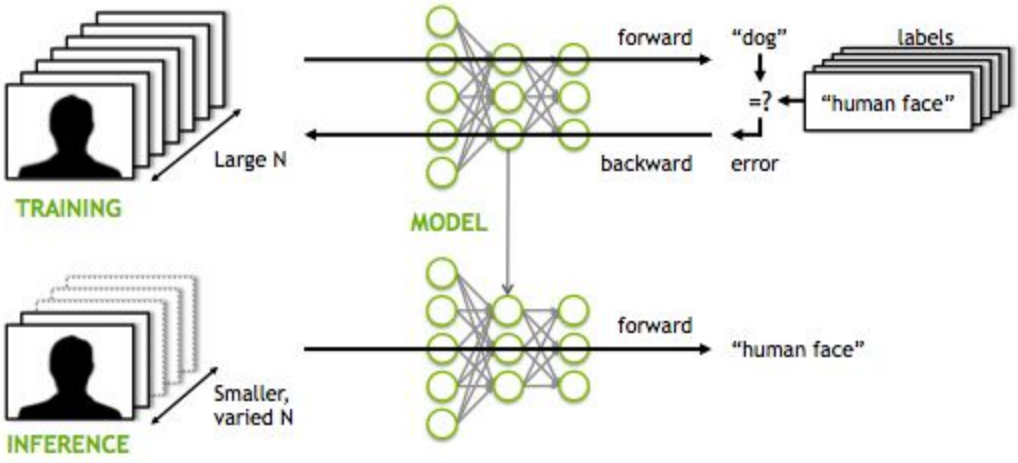
CUDA Parallel Programming Framework

The CUDA parallel programming framework from NVIDIA is a particular implementation of the GPGPU paradigm. CUDA once was an acronym for Compute Unified Device Architecture, but NVIDIA dropped the expansion and now just uses CUDA. This architecture, facilitating our machine learning parallelization via GPU acceleration (another way to refer to GPGPU), requires particular consideration in order to effectively manage available resources and provide the maximum execution speed benefit.

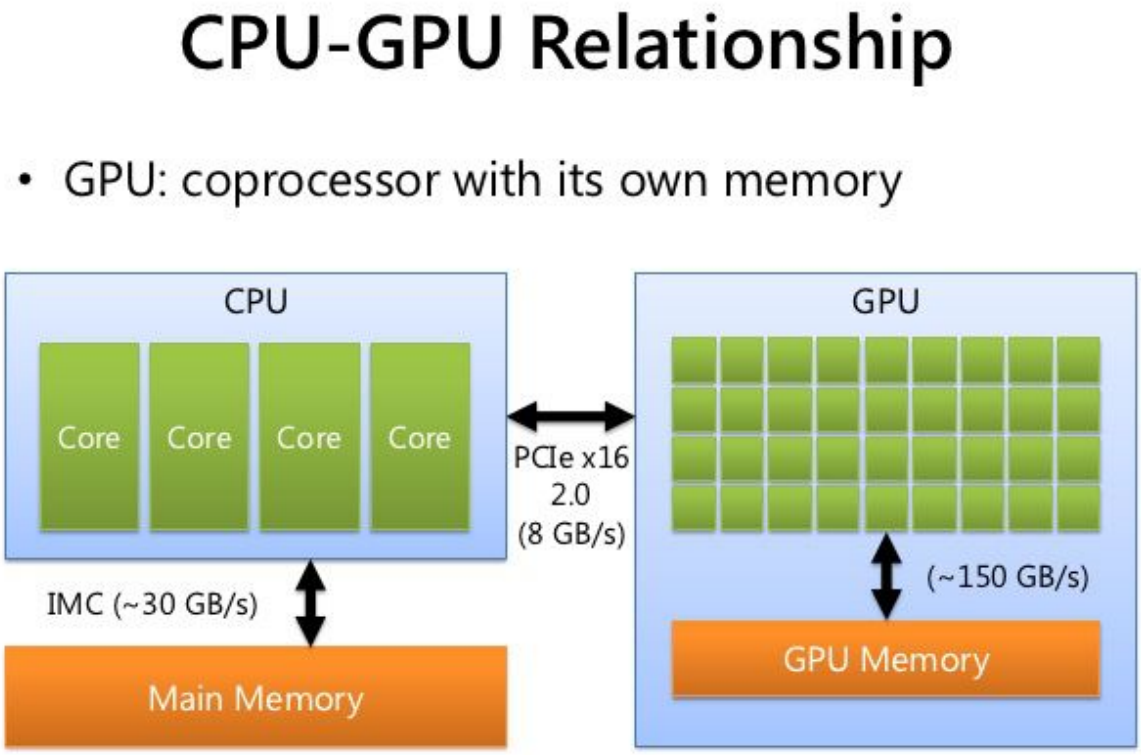
CUDA is technically a heterogeneous computing environment, meaning that it facilitates coordinated computing on both CPUs and GPUs. The CUDA architecture consists of hosts and devices, with **host** referring to a traditional CPU, and **device** referencing processors with large numbers of arithmetic units, typically GPUs. CUDA provides extensions to traditional programming languages

(the native CUDA bindings are C, but have been ported or made otherwise available to many additional languages), enabling the creation of **kernels**, which are parallel-executing functions.

A kernel, when launched, gets simultaneously executed by a large number of CUDA device **threads**, a collection of which are referred to as a **block** of threads, blocks being collected into **grids**. Threads are arranged in 3-dimensional layouts within blocks, which are, in turn, arranged in 3-dimensional layouts within grids. Figure 4 demonstrates these relationships and layouts. The total number of threads, blocks, and grids employed by a particular kernel are strategically dictated by a programmer's code executing on the host at kernel launch, based on given situational requirements.



Importantly, hosts and devices possess their own memory spaces, independent of one another. A CUDA device shares a single global memory space. The first requirement for launching kernels and spawning numerous device threads for computation is to copy the required data from host to device memory. Once computation is complete, it is necessary to copy the results back in the reverse direction. This is all facilitated via CUDA extensions, and occurs at a heavily abstracted layer from the programmer's point of view.



When managing device-side memory, proper allocation of blocks to kernels is critical; too few leads to a lack of computational power, while too many results in wasted threads, which could have been assigned to other simultaneously-executing kernels. As a specific example, this could translate to too few threads allocated to a particular fold during k-fold cross-validation model-building, leading to a much longer validation process than intended.

Conversely, it could result in too many threads being assigned during k-fold cross-validation model building, leaving large numbers of unused threads and extending the amount of time required for all folds to complete their model validation.

Fortunately, the management of device memory, including the number of threads assigned to blocks, and, ultimately, to kernels, is user-definable (within some upper bounds, such as a maximum of 1024 threads per block). CUDA provides some clever ways of semi-automation of this management as well, allowing memory management functions to take mathematical expressions as arguments, so that, for example, a kernel can, upon execution, calculate the size of a data structure such as an array or a matrix and allocate the amount and dimensions of memory that would be appropriate for its computations.

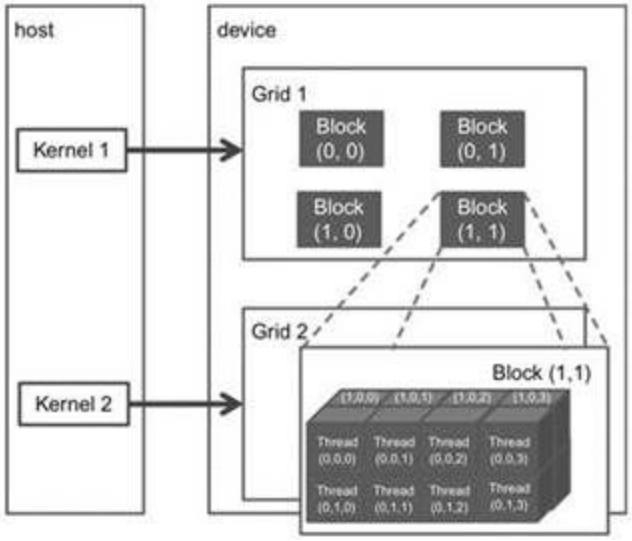


Fig. 4: CUDA Grid Organization.

Consider matrix multiplication, an aspect of linear regression which we propose to parallelize, and its implementation on the CUDA architecture. Proceeding at a high level, without regard to matrix sizes, we can say that we have 2 matrices to multiply, M and N, and that the result will be stored in matrix P. First, we allocate space for matrices M and N in device global memory, as well as space for the resulting matrix P. We then copy matrices M and N to the device.

Assuming for simplicity that all matrices fit into a single block, we will have each block thread compute an element of P. To accomplish this, each thread loads a row of M and a column of N, computes the dot product, and stores it in the appropriate element of P. As each of these dot products are computed in parallel, the total time it will take to perform the matrix multiplication is the time that it takes to perform a single dot product computation. Once complete, matrix P is then copied from device memory back to host memory, where it can be further used by serial code, if necessary. Typically such a kernel operation would be followed by deallocation of device memory.

This is a high level overview. In practice, additional tasks need to be performed, such as determining block sizes, as stated above. It is also a single, specific example; however, the memory management and device computation techniques, while they will be, by necessity, quite different by algorithmic situation, generalize to our various tasks: identify parallelizable computations, allocate device memory, copy data to device, perform parallelized computation, copy result back to host, continue with serial code.

Algorithmic Applications in Machine Learning

Given the proper data, knowledge of algorithm implementation, and ambition, there is no limit to what you can attempt with parallel processing in machine learning. Of course, and as mentioned above, identifying parallelizable portions of code is the most difficult task, and there may not be any in a given algorithm.

A good place to start is matrix multiplication, as treated above, which is a well-used method for implementing linear regression algorithms. An implementation of linear regression on GPU can be found here. The paper "Performance Improvement of Data Mining in Weka through GPU Acceleration" notes speed increases, and the paper provides some additional insight into conceptualizing parallelism algorithmically.

Another common task used in machine learning which is ripe for parallelization is distance calculation. Euclidean distance is a very common metric which requires calculation over and over again in numerous algorithms, including k-means clustering. Since the individual distance calculations of successive iterations are not dependent on other calculations of the same iteration, these calculations could be performed in parallel (if we forget our memory management overhead as a potential bottleneck to contend with).

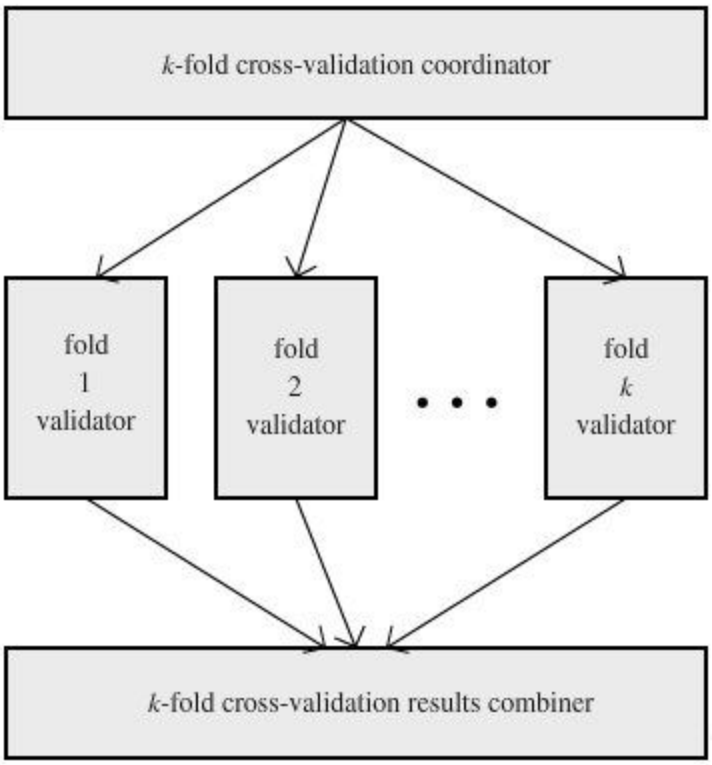


Fig. 5: k-fold Cross-validation.

While these aforementioned shared statistical tasks could benefit from efficiency of execution, there is an additional aspect of the machine learning data flow which could potentially allow for even more significant gains. A common evaluation technique regularly employed in machine learning model validation is k-fold cross-validation, involving the intensive, not-necessarily sequential processing of dataset segments. k-fold cross-validation is a deterministic method for model building, achieved by leaving out one of k-segments, or folds, of a dataset, training on all k-1 segments, and using the remaining kth segment for testing; this process is then repeated k times, with the individual prediction error results being combined and averaged in a single, integrated model. This provides variability, with the goal of producing the most accurate predictive models possible.

This extensive model validation, when performed sequentially, can be relatively time-consuming, especially when each fold is paired with a computationally expensive algorithm task such as linear regression matrix multiplication. As k-fold cross-validation is a standard method for predicting a given machine learning algorithm's error rate, attempting to increase the speed by which this prediction occurs seems particularly worthy of effort. A very high level view of doing so is implied previously in this article.

A consideration for those using Python goes beyond algorithm design, and relates to optimized native code and runtime comparisons with parallel implementations. While beyond the scope of this discussion, you may want to read more about this topic here.

Thinking algorithmically is necessary to leverage finite computational resources in any situation, and is no different with machine learning. With some clever thinking, an in-depth understanding of what you are attempting, and a collection of tools and their documentation, you never know what you may be able to achieve. Trust me when I say that, after doing some related work in grad school, finding opportunities to experiment with take much less time than you might think. Familiarize yourself with a code base, read a few tutorials, and get to work.

Parallel computing, GPUs, and traditional machine learning can be good friends, and I challenge you to dig deeper and discover the potential for yourself.

http://www.kdnuggets.com/2016/11/parallelism-machine-learning-gpu-cuda-threading.htm

Accelerate Machine Learning with the cuDNN Deep Neural Network Library

NVIDIA cuDNN is a GPU-accelerated library of primitives for DNNs. It provides tuned implementations of routines that arise frequently in DNN applications, such as:

- convolution
- pooling
- softmax
- neuron activations, including:
    - ➢ Sigmoid
    - ➢ Rectified linear (ReLU)
    - ➢ Hyperbolic tangent (TANH)

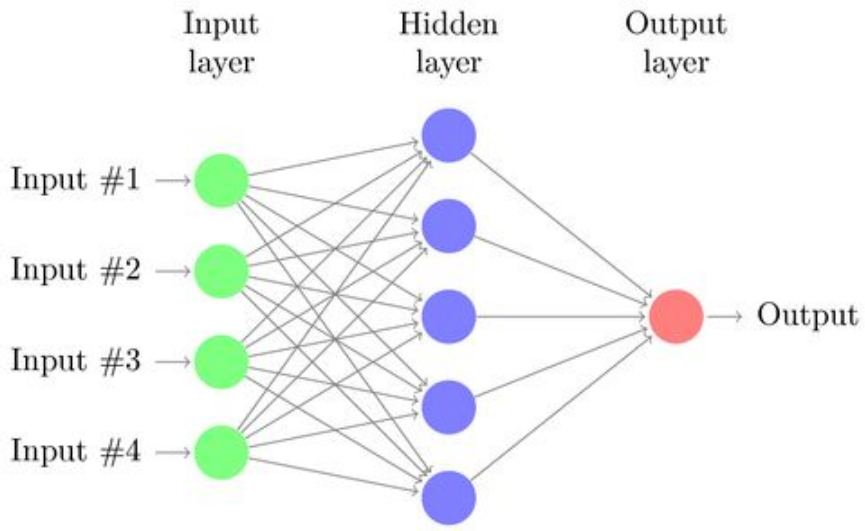
Of course these functions all support the usual forward and backward passes. cuDNN's convolution routines aim for performance competitive with the fastest GEMM-based (matrix multiply) implementations of such routines while using significantly less memory.

cuDNN features customizable data layouts, supporting flexible dimension ordering, striding and subregions for the 4D tensors used as inputs and outputs to all of its routines. This flexibility allows easy integration into any neural net implementation and avoids the input/output transposition steps sometimes necessary with GEMM-based convolutions.

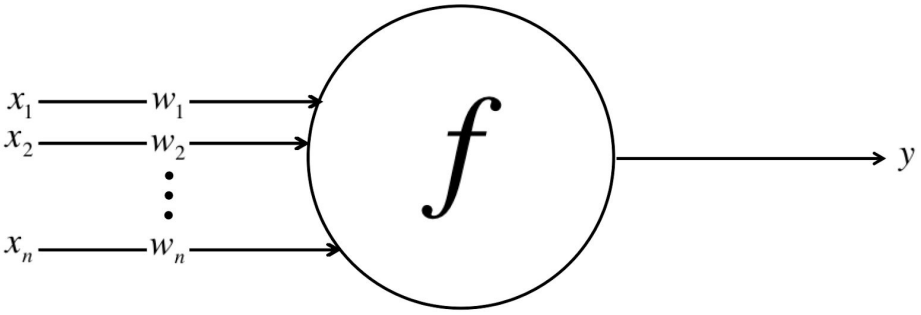
cuDNN is thread safe, and offers a context-based API that allows for easy multithreading and (optional) interoperability with CUDA streams. This allows the developer to explicitly control the library setup when using multiple host threads and multiple GPUs, and ensure that a particular GPU device is always used in a particular host thread (for example).

cuDNN allows DNN developers to easily harness state-of-the-art performance and focus on their application and the machine learning questions, without having to write custom code. cuDNN works on Windows or Linux OSes, and across the full range of NVIDIA GPUs, from low-power embedded GPUs like Tegra K1 to high-end server GPUs like Tesla K40. When a developer leverages cuDNN, they can rest assured of reliable high performance on current and future NVIDIA GPUs, and benefit from new GPU features and capabilities in the future.

A First Look at Neural Networks

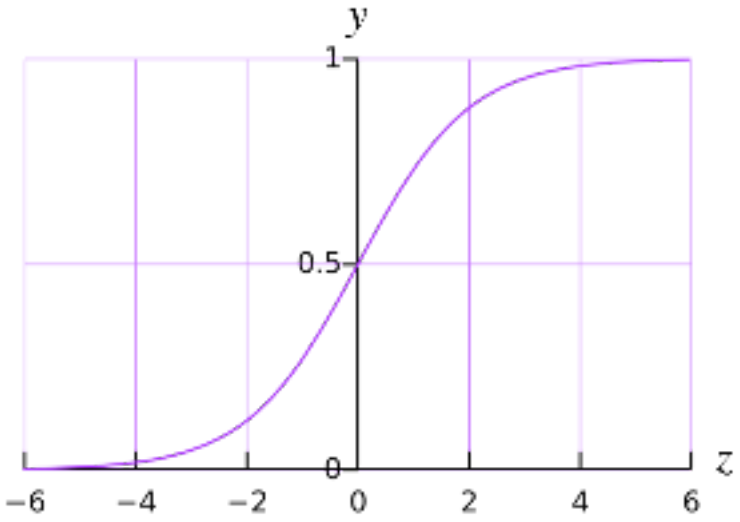


Deep learning is a form of machine learning that uses a model of computing that's very much inspired by the structure of the brain. Hence we call this model a neural network. The basic foundational unit of a neural network is the neuron, which is actually conceptually quite simple.



Schematic for a neuron in a neural net

Each neuron has a set of inputs, each of which is given a specific weight. The neuron computes some function on these weighted inputs. A linear neuron takes a linear combination of the weighted inputs. A sigmoidal neuron does something a little more complicated:



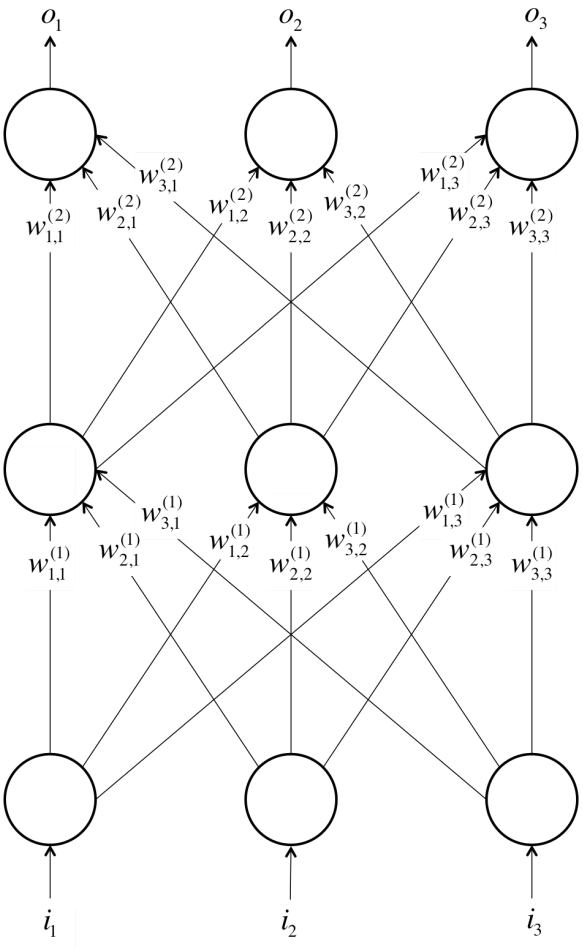
$$z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$

$$y = \frac{1}{1 + e^{-z}}$$

The function of a sigmoidal neuron

It feeds the weighted sum of the inputs into the logistic function. The logistic function returns a value between 0 and 1. When the weighted sum is very negative, the return value is very close to 0. When the weighted sum is very large and positive, the return value is very close to 1. For the more mathematically inclined, the logistic function is a good choice because it has a nice looking derivative, which makes learning a simpler process. But technical details aside, whatever function the neuron uses, the value it computes is transmitted to other neurons as its output. In practice, sigmoidal neurons are used much more often than linear neurons because they enable much more versatile learning algorithms compared to linear neurons.

A neural network comes about when we start hooking up neurons to each other, to the input data, and to the "outlets," which correspond to the network's answer to the learning problem. To make this structure easier to visualize, I've included a simple example of a neural net below. We let w(k)i,j be the weight of the link connecting the ith neuron in the kth layer with the jth neuron in the k+1st layer:



An example of a neural net with 3 layers and 3 neurons per layer

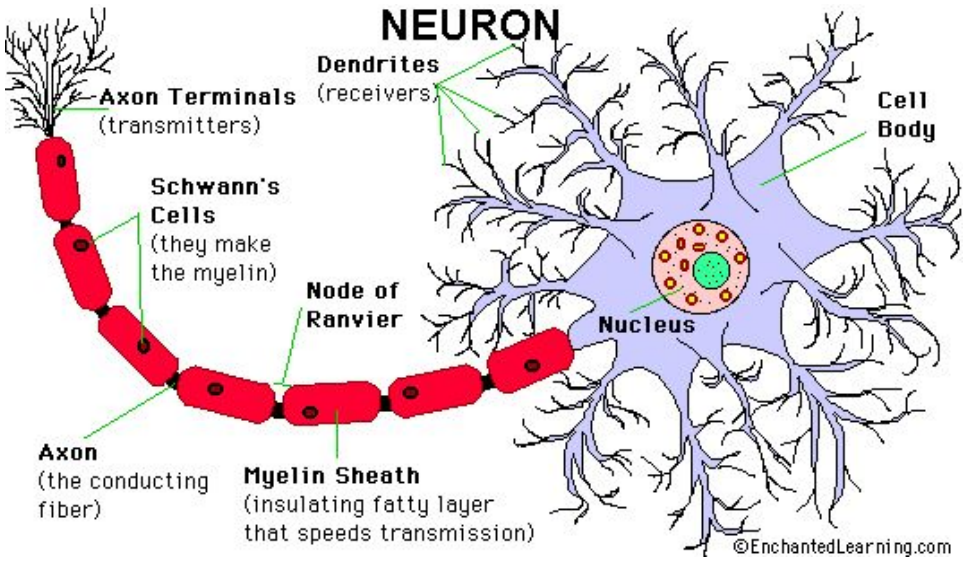
Similar to how neurons are generally organized in layers in the human brain, neurons in neural nets are often organized in layers as well, where neurons on the bottom layer receive signals from the inputs, where neurons in the top layers have their outlets connected to the "answer," and where are usually no connections between neurons in the same layer (although this is an optional restriction, more complex connectivities require more involved mathematical analysis). We also note that in this example, there are no connections that lead from a neuron in a higher layer to a neuron in a lower layer (i.e., no directed cycles). These neural networks are called *feed-forward* neural networks as opposed to their counterparts, which are called *recursive* neural networks (again these are much more complicated to analyze and train). For the sake of simplicity, we focus only on feed-forward networks throughout this discussion. Here's a set of some more important notes to keep in mind:

1) Although every layer has the same number of neurons in this example, this is not necessary.

2) It is not required that a neuron has its outlet connected to the inputs of every neuron in the next layer. In fact, selecting which neurons to connect to which other neurons in the next layer is an art that comes from experience. Allowing maximal connectivity will more often than not result in *overfitting*, a concept which we will discuss in more depth later.

3) The inputs and outputs are *vectorized* representations. For example, you might imagine a neural network where the inputs are the individual pixel RGB values in an image represented as a vector. The last layer might have 2 neurons which correspond to the answer to our problem: [0,1] if the image contains a dog, [1,0] if the image contains a cat, [0,0] if it contains neither, and [1,1] if it contains both.

4) The layers of neurons that lie sandwiched between the first layer of neurons (input layer) and the last layer of neurons (output layer), are called *hidden layers*. This is because this is where most of the magic is happening when the neural net tries to solve problems. Taking a look at the activities of hidden layers can tell you a lot about the features the network has learned to extract from the data.



Training a Single Neuron

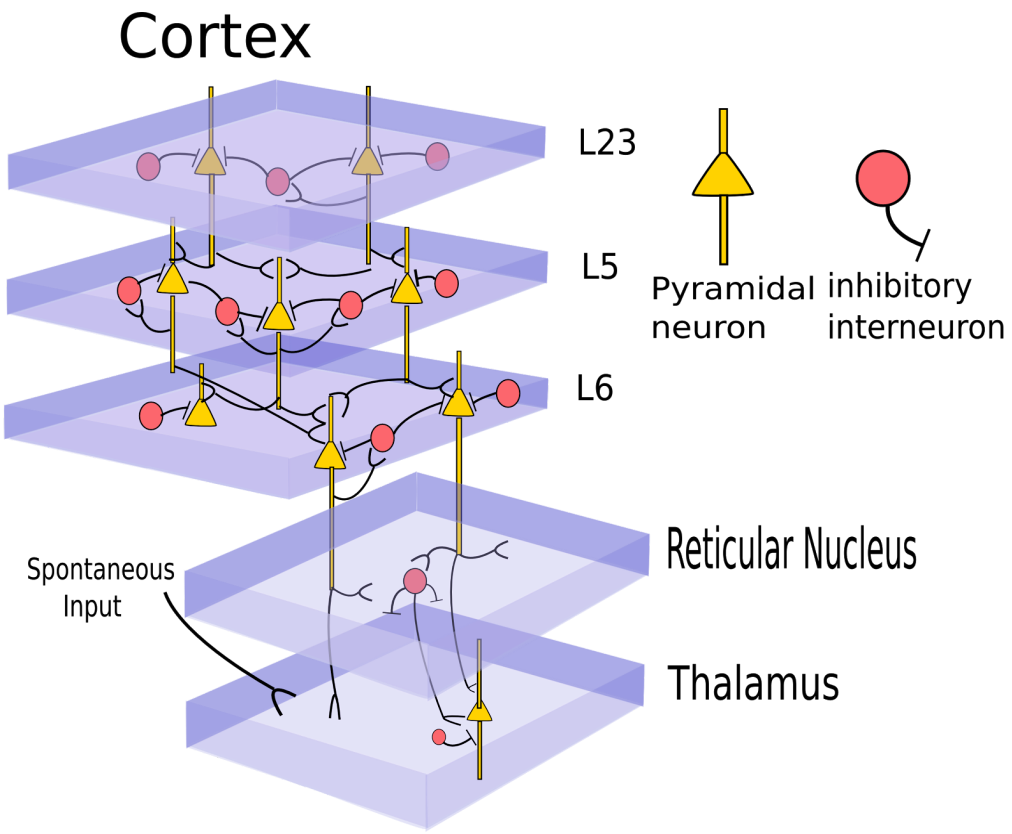
Well okay, things are starting to get interesting, but we're still missing a big chunk of the picture. We know how a neural net can compute answers from inputs, but we've been assuming that we know what weights to use to begin with. Finding out what those weights should be is the hard part of the problem, and that's done through a process called *training*. During training, we show the neural net a large number of training examples and iteratively modify the weights to minimize the errors we make on the training examples.

Let's start off with a toy example involving a single linear neuron to motivate the process. Every day you grab lunch in the dining hall where your meal consists completely of burgers, fries, and soda.

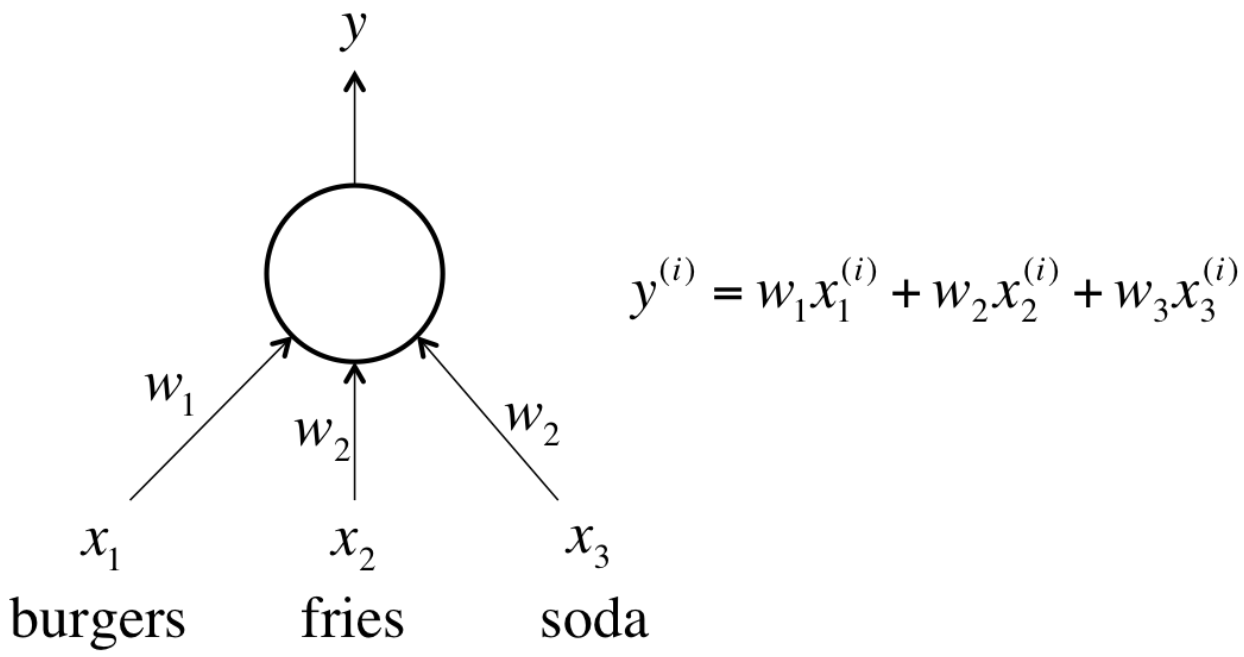
You buy some number of servings of each item. You want to be able to predict how much your meal will cost you, but you don't know the prices of each individual item. The only thing the cashier will tell you is the total price of the meal.

How do we solve this problem? Well, we could begin by being smart about picking our training cases, right? For one meal we could buy only a single serving of burgers, for another we could only buy a single serving of fries, and then for our last meal we could buy a single serving of soda. In general, choosing smart training cases is a very good idea. There's lots of research that shows that by engineering a clever training set, you can make your neural net a lot more effective. The issue with this approach is that in real situations, this rarely ever gets you even 10% of the way to the solution. For example, what's the analog of this strategy in image recognition?

Brain Neurons and Spontaneous Input



Let's try to motivate a solution that works in general. Take a look at the single neuron we want to train:



The neuron we want to train for the Dining Hall Problem

Let's say we have a bunch of training examples. Then we can calculate what the neural network will output on the *ith* training example using the simple formula in the diagram. We want to train the neuron so that we pick the optimal weights possible - the weights that minimize the errors we make on the training examples. In this case, let's say we want to minimize the square error over all of the training examples that we encounter. More formally, if we know that t(i) is the true answer for the *ith* training example and y(i) is the value computed by the neural network, we want to minimize the value of the error function E:

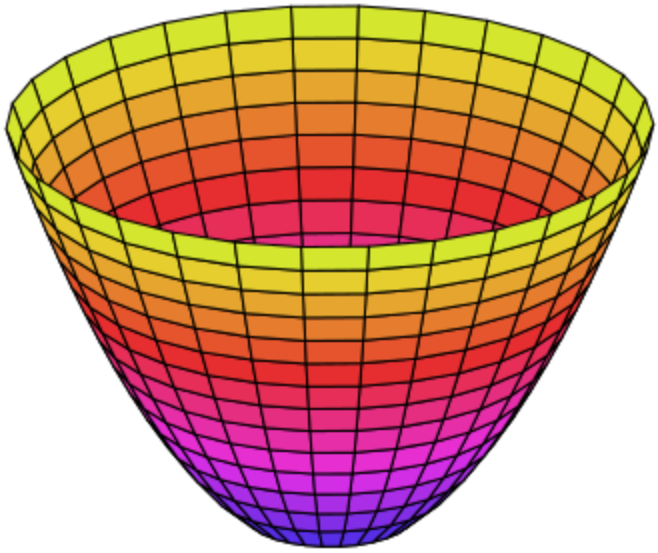
$$E = \frac{1}{2} \sum_i (t^{(i)} - y^{(i)})^2$$

Now at this point you might be thinking, wait up... Why do we need to bother ourselves with this error function nonsense when we have a bunch of variables (weights) and we have a set of equations (one for each training example)? Couldn't we just solve this problem by setting up a system of linear system of equations? That would automatically give us an error of zero assuming that we have a consistent set of training examples, right?

That's a smart observation, but the insight unfortunately doesn't generalize well. Remember that although we're using a linear neuron here, linear neurons aren't used very much in practice because

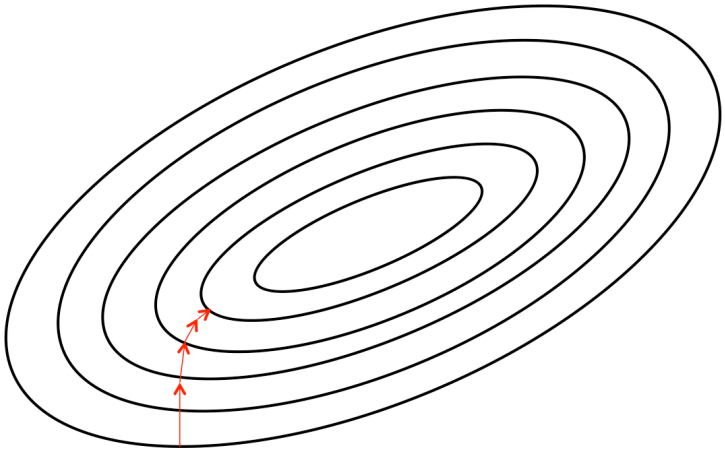
they're constrained in what they can learn. And the moment you start using nonlinear neurons like the sigmoidal neurons we talked about, we can no longer set up a system of linear equations!

So maybe we can use an iterative approach instead that generalizes to nonlinear examples. Let's try to visualize how we might minimize the squared error over all of the training examples by simplifying the problem. Let's say we're dealing with a linear neuron with only two inputs (and thus only two weights, w1 and w2). Then we can imagine a 3-dimensional space where the horizontal dimensions correspond to the weights w1 and w2, and there is one vertical dimension that corresponds to the value of the error function E. So in this space, points in the horizontal plane correspond to different settings of the weights, and the height at those points corresponds to the error that we're incurring, summed over all training cases. If we consider the errors we make over all possible weights, we get a surface in this 3-dimensional space, in particular a quadratic bowl:



The quadratic error surface for a linear neuron

We can also conveniently visualize this surface as a set of elliptical contours, where the minimum error is at the center of the ellipses:



Visualizing the error surface as a set of contours

So now let's say we find ourselves somewhere on the horizontal plane (by picking a random initialization for the weights). How would we get ourselves to the point on the horizontal plane with the smallest error value? One strategy is to always move perpendicularly to the contour lines. Take a look, for instance, at the path denoted by the red arrows. Quite clearly, you can see that following this strategy will eventually get us to the point of minimum error.

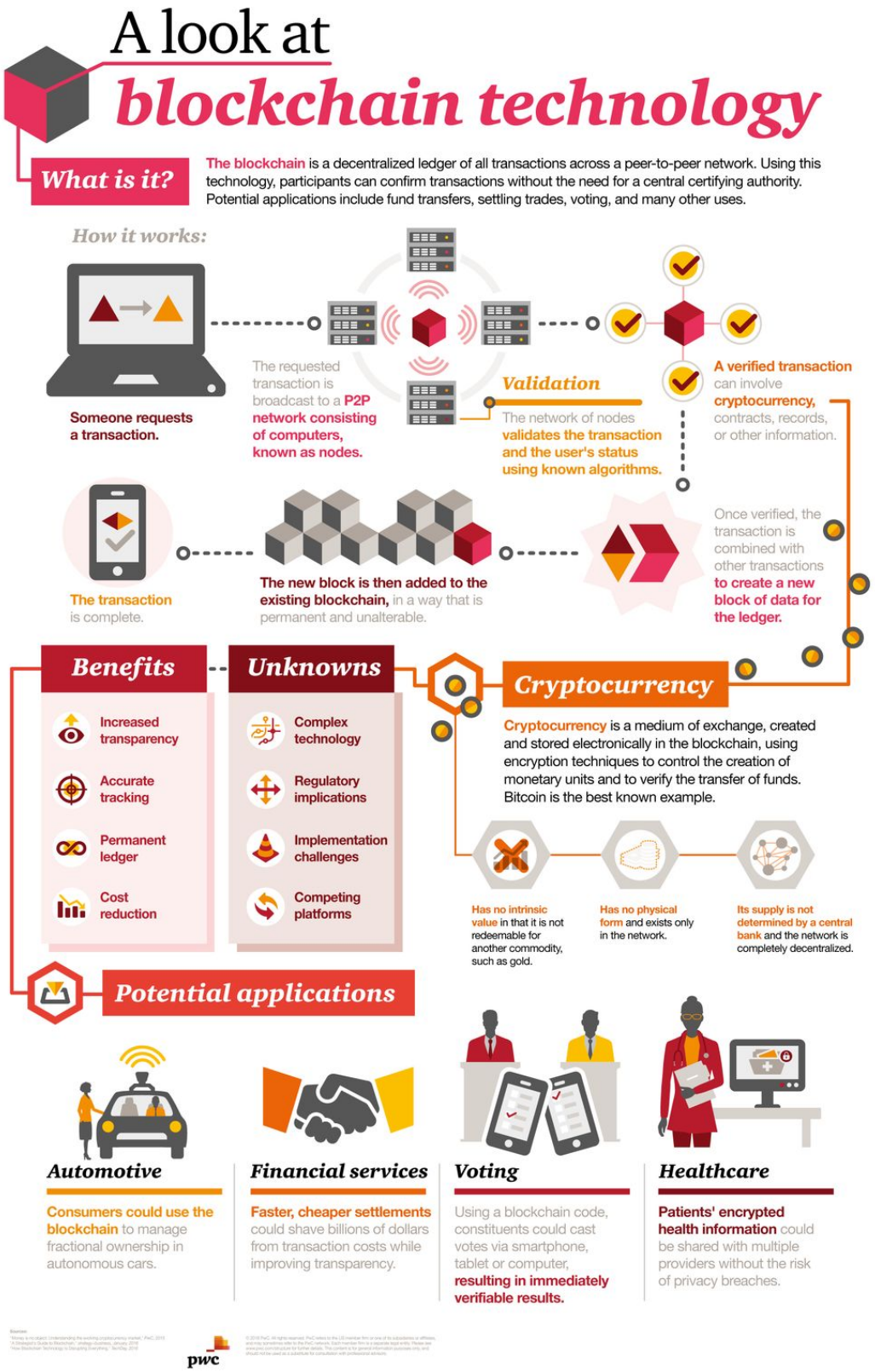
What's particularly interesting is that moving perpendicularly to the contour lines is equivalent to taking the path of steepest descent down the parabolic bowl. This is a pretty amazing result from calculus, and it gives us the name of this general strategy for training neural nets: *gradient descent*.

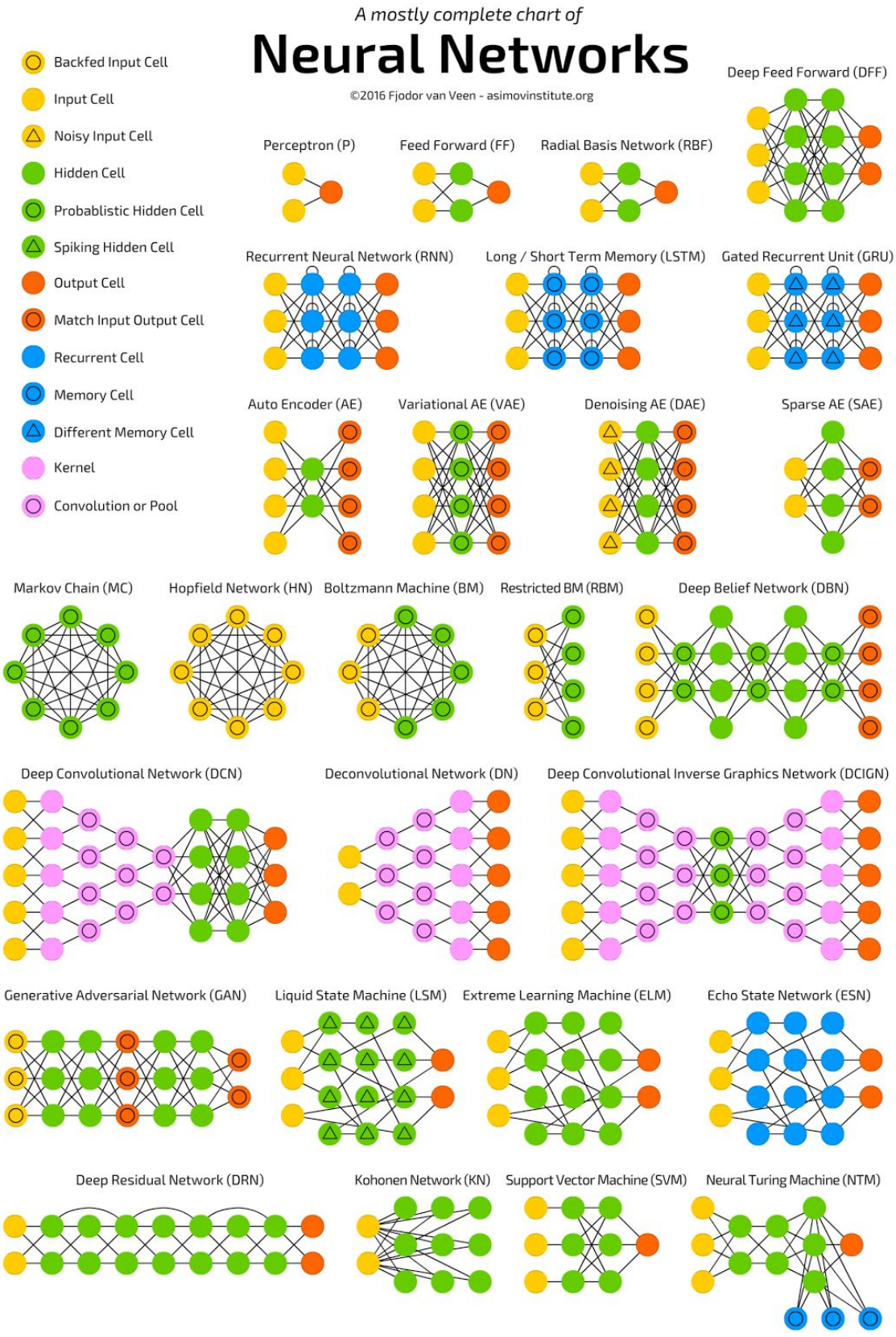
http://www.kdnuggets.com/2015/01/deep-learning-explanation-what-how-why.html

Why Blockchain?

The advantage of blockchain is that it is decentralized – no single person or company controls data entry or its integrity; however, the sanctity of the blockchain is verified continuously by every computer on the network. As all points hold the same information, corrupt data at point “A” can’t become part of the chain because it won’t match up with the equivalent data at points “B” and “C”.

With the above in mind, blockchain is immutable – information remains in the same state for as long as the network exists.


A look at
blockchain technology
What is it?
The blockchain is a decentralized ledger of all transactions across a peer-to-peer network. Using this technology, participants can confirm transactions without the need for a central certifying authority. Potential applications include fund transfers, settling trades, voting, and many other uses.
How it works:
Someone requests a transaction.
The requested transaction is broadcast to a P2P network consisting of computers, known as nodes.
Validation
The network of nodes validates the transaction and the user's status using known algorithms.
A verified transaction can involve cryptocurrency, contracts, records, or other information.
The transaction is complete.
The new block is then added to the existing blockchain, in a way that is permanent and unalterable.
Once verified, the transaction is combined with other transactions to create a new block of data for the ledger.
Benefits
Increased transparency
Accurate tracking
Permanent ledger
Cost reduction
Unknowns
Complex technology
Regulatory implications
Implementation challenges
Competing platforms
Cryptocurrency
Cryptocurrency is a medium of exchange, created and stored electronically in the blockchain, using encryption techniques to control the creation of monetary units and to verify the transfer of funds. Bitcoin is the best known example.
Has no intrinsic value in that it is not redeemable for another commodity, such as gold.
Has no physical form and exists only in the network.
Its supply is not determined by a central bank and the network is completely decentralized.
Potential applications
Automotive
Consumers could use the blockchain to manage fractional ownership in autonomous cars.
Financial services
Faster, cheaper settlements could shave billions of dollars from transaction costs while improving transparency.
Voting
Using a blockchain code, constituents could cast votes via smartphone, tablet or computer, resulting in immediately verifiable results.
Healthcare
Patients' encrypted health information could be shared with multiple providers without the risk of privacy breaches.
pwc

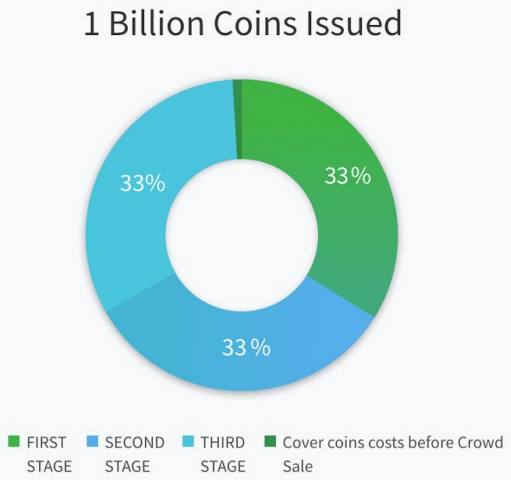

A mostly complete chart of
Neural Networks
©2016 Fjodor van Veen - asimovinstitute.org
Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Deep Feed Forward (DFF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Kohonen Network (KN)
Support Vector Machine (SVM)
Neural Turing Machine (NTM)

CROWD SALE COIN DISTRIBUTION

We have organized a crowd sale where you can purchase our tokens, which are the internal means of calculating the **Neural.Club**. These coins will be the main asset within the system, they can be spent for data analysis or sell.

The power of the network will be provided by the capacities of the miners due to their video cards and other equipment. Instead, a reward will be established based on the complexity of the necessary analysis and the amount of incoming data.

- **PRESALE STAGE** - Sell **1%**. Phase for determine the actual cost of Crowdsale.
- **First Crowdsale STAGE** - 20-30 days. Sell **33%** at a fixed price after presale.
- **Second Crowdsale STAGE** - 20-30 days. Sell **33%** at **First STAGE** Crowdsale **+ 10%**
- **Third Crowdsale STAGE** - 20-30 days. Sell **33%** at **Second STAGE** Crowdsale **+ 10%**



Total coins supply: **1 billion (1,000,000,000) NEURAL**, of which:

- **33%** sold in the coins crowd sale STAGE 1
- **33%** sold in the coins crowd sale STAGE 2
- **33%** sold in the coins crowd sale STAGE 3
- **1%** to cover crowdsale coins costs

Purchase methods accepted: BTC and ETH

The funding raised through the crowd sale will be used for development purposes, set up factory and local warehouses, advancement of the products and as working capital.

Today we are on the verge of a new discovery. Unknown, but great. We invite everyone to participate in the opening, which will certainly go down in history.

Disclaimer

NEITHER THE SOFTWARE NOR ITS CREATORS PROVIDE LEGAL ADVICE AND THIS CODE WAS NOT CREATED TO PROVIDE LEGAL ADVICE OR AS A SUBSTITUTE FOR LEGAL ADVICE. BY USING THIS CODE YOU ALSO AGREE:

The creators of the Software and its contributors are not your lawyers.

The Software is not a lawyer.

Your use of the Software does not, in and of itself, create a legally binding contract in any jurisdiction and does not establish a lawyer-client relationship. Your communication with a non-lawyer will not be subject to the attorney-client privilege and (depending on your jurisdiction) may not be entitled to protection as confidential communication.

The dissemination, distribution, or usage of this software shall not constitute the provision of legal advice within your jurisdiction. Unless you are legally authorized and licensed to do so, you will not use the Software to provide or assist in the provision of legal advice.

You acknowledge and understand that each jurisdiction has its own particular rules regarding the practice of law. IF YOU USE THIS SOFTWARE TO PROVIDE LEGAL ADVICE YOU MAY BE SUBJECT TO CIVIL AND CRIMINAL LIABILITY. PRACTICING LAW WITHOUT A LICENSE IS A VIOLATION OF CRIMINAL LAW IN SOME JURISDICTIONS. CONSULT A LAWYER LICENSED IN YOUR JURISDICTION IF YOU HAVE ANY QUESTIONS ABOUT WHAT DOES OR DOES NOT CONSTITUTE THE PRACTICE OF LAW.

The providers of this software neither warrant nor guarantee this software shall meet the requirements of any particular legal system to form a legally binding contract, nor it their intention to directly or indirectly facilitate or encourage the unauthorized practice of law.

You agree that in order for you to form a legally binding contract that you shall seek legal advice from an appropriately qualified and experienced lawyer within your jurisdiction.

Minting of tokens may constitute the sale of securities in certain jurisdictions. Seek appropriate legal advice before deploying a crowdsale contract.

We are currently still under construction but feel free to look around!

JOIN US

LINKS

WEB	WWW.NEURAL.CLUB
TWITTER :	https://twitter.com/Neural_Club
Facebook:	http://facebook.com/neural.club.7
KeyBASE:	https://keybase.io/neuralclub
GITHUB:	http://github.com/neuralclub
YouTube:	youtube.com/channel/UCQxwaoN7WvIUOIkxUk72RAA

Projects

neural.accountant

neural.club

neural.training

neural.money

neural.cash

neural.partners

neural.wiki

neural.team

neural.watch

neural.show

Keywords

- Machine learning
- Artificial intelligence
- Data science
- Predictive analytics
- Blockchain microfinances
- Distribution Decentralized
- Nvidia[tm] CUDA[tm] & cuDNN
- Google[tm] Tensorflow[tm]
- API integration
- Science mining
- Data mining
- GUI & SDK

2017