Chapter on Machine Learning

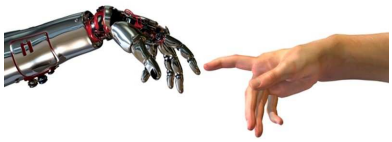
By Jeff Edmonds

Computers can now drive cars and find cancer in x-rays. For better or worse, this will change the world (and the job market). Strangely designing these algorithms is not done by telling the computer what to do or even by understanding what the computer does. The computers learn themselves from lots and lots of data and lots of trial and error. This learning process is more analogous to how brains evolved over billions of years of learning. The machine itself is a neural network which models both the brain and silicon and-or-not circuits, both of which are great for computing. The only difference with neural networks is that what they compute is determined by weights and small changes in these weights give you small changes in the result of the computation. The process for finding an optimal setting of these weights is analogous to finding the bottom of a valley. "Gradient Decent" achieves this by using the local slope of the hill (derivatives) to direct the travel down the hill, i.e. small changes to the weights. There is some theory. If a machine is found that gives the correct answers on the randomly chosen training data without simply memorizing, then we can prove that with high probability this same machine will also work well on never seen before instances.

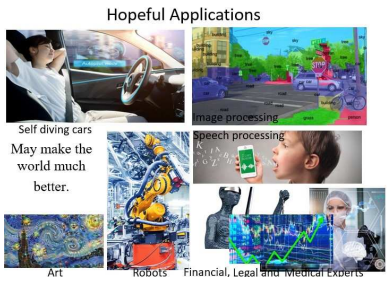
Coding: When writing computer code, the instructions are painstakingly written by humans. This works great for simple repetitive tasks. Expert systems of the 80's however failed. Though we are able to walk, it is hard to explaining how to do it!!!

Machine Learning: When designing neural networks, the instructions are 100% dictated by weights $\langle w_1, \ldots, w_m \rangle$. Like the brain, these are way too hard to understand. They are learned using a simple algorithm, finding patterns in lots of data. Walking, for example, has enough of a pattern that machine learning can copy it.

Evolution: Add just a little random change that is encouraged by some feed back to go in the right direction and the *Emergent Complexity* that arises is awe inspiring. I find the parallels between evolution and machine learning oddly spiritual.



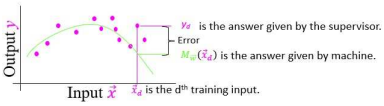
Hopeful Applications: Self diving cars, image processing, speech processing, robots, art, medical, financial, and legal experts. Hopefully these will make the world a better place.



Scary Applications: People losing their jobs, the system watching you, ads customized to you, cybersecurity picking out "criminals" in a crowd, killing machines, machines deciding your fate. These might make the world a worse place.

Abstract Thinking: We can talk about many aspects of machine learning without needing to know the nature of the machine or the nature of the data. We talk of knowing if image is a face without mentioning noses. This simplicity let's us focus on what is important and ensures that what we say works for any model of machine and for any computational problem.

Supervised Training Data: The input that we receive is a set of input/output pairs $\{\langle \vec{x}_1, y_1 \rangle, \langle \vec{x}_2, y_2 \rangle, \ldots, \langle \vec{x}_D, y_D \rangle\}$. In order to be able to draw a simple graph for intuitive purposes, we pretend each input $\vec{x}_d$ is a real numbers. But more likely it is something complex like an image. For each such input, y_d is the answer provided by a *Supervisor*. When y_d is a single real number, we call the process *regression*. When y_d is a label like cat or dog, we call the process *categorizing*. Another way of visualizing the data is to assume that each input $\vec{x}_d$ is some point in some height dimensional space and the supervisor's answer y_s is indicated by the colour of the point.



Cat: If the input $\vec{x}_d$ is an image of a cat, then the computer just sees a big matrix of numbers. From this, meaning must be extracted. This seems to me to be magic. On the other hand, the brain manages to do it.

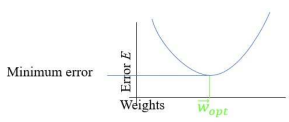
Machine: Our goal is to build a machine $M_{\vec{w}}(\vec{x})$ that takes an input $\vec{x}$ and returns an answer y. It is parametrized by a vector of m real valued weights $\vec{w} = \langle w_1, \ldots, w_m \rangle$. This includes anything that the learning process learns and remembers about the data and uses later to make predictions.

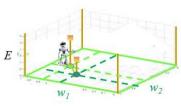
Linear and Non-Linear Regression: For example, y_d might be the likelihood of rain on day d, $x_{\langle d,1 \rangle}$ whether there are clouds, $x_{\langle d,2 \rangle}$ which the colour of coat the man is wearing, and $\langle w_1, w_2 \rangle$ their level of effect on rain. Our machine might might then approximate y_d with $M_{\vec{w}}(\vec{x}_d) = w_0 + w_1 x_{\langle d,1 \rangle} + w_2 x_{\langle d,2 \rangle}$. Who knows, maybe if people are wearing yellow coats, then these might be rain coats and hence it is more likely to rain. The model designer could make the types of machines learned more powerful by adding additional terms like $+w_3\, {x_{\langle d,2 \rangle}}^2$. Here ${x_{\langle d,2 \rangle}}^2$ is a hand picked precomputed features of the input $\vec{x}_d$ and w_3 is its weight that is learned during the learning process. Though this machine is now non-linear in the input $\vec{x}_d$, we still call this linear regression, because the answer is linear in the learned weights w_3. In contrast, learning $M_{\vec{w}}(\vec{x}_d) = {x_{\langle d,1 \rangle}}^{w_1} + {x_{\langle d,2 \rangle}}^{w_2}$, being non-linear in the weights, would be much harder.

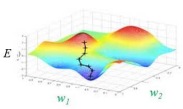
Neural Networks: Our machine might also be something much more complex like a *neural network*. All you need to know now about it now is that how it computes $y = M_{\vec{w}}(\vec{x})$ is dictated by its weights $\vec{w} = \langle w_1, \ldots, w_m \rangle$.

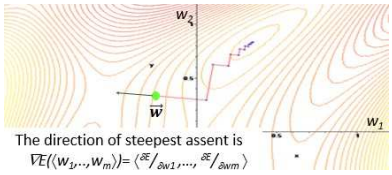
Error: Given some setting $\vec{w} = \langle w_1, \ldots, w_m \rangle$ of the weights, we want to measure how well the machine $M_{\vec{w}}(\vec{x})$ does at getting the correct answers y on the training data $\{\langle \vec{x}_1, y_1 \rangle, \langle \vec{x}_2, y_2 \rangle, \ldots, \langle \vec{x}_D, y_D \rangle\}$. The easiest thing to do would be to count the fraction of the data set for which the machine gets the correct answer, namely $y_d = M_{\vec{w}}(\vec{x}_d)$. However, this gives little feedback on how to improve the machine. If answer y is a real number, the difference $y_d - M_{\vec{w}}(\vec{x}_d)$ gives us a measure of how close our machine is to the correct answer. We could take the absolute value of this $|y_d - M_{\vec{w}}(\vec{x}_d)|$ because we do not care which direction the error is in, but $|z|$ is too pointy at $z = 0$ making the math near impossible. Instead, we square it and sum over all data items, namely $E(\vec{w}) = \sum_d (y_d - M_{\vec{w}}(\vec{x}_d))^2$. When the answer y is a category like "cat", we can have the machine instead produce a real number telling us the "probability" that it thinks it is a "cat". The error would then be based on how close the probability of the correct answer is to one.

Machine Learning: Given the training data, our goal is to find the weights $\vec{w}_{opt}$ that minimizes the error $E(\vec{w})$. This is all that machine learning is.







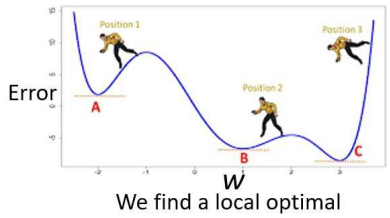


Error Surface: We think of the error as a function of the weights $\vec{w}$ because they are the parameters that we are able to tweak. If we pretend that the weights consist of one real number $\vec{w} = \langle w_1 \rangle$, then we can graph the error function on paper with w_1 being the x-axis and the error $E(\vec{w})$ being the y-axis. We are looking for a minimum. With two real numbers $\vec{w} = \langle w_1, w_2 \rangle$, we get an error surface. The first weight w_1 can tell you your East-West location, the second w_2 your North-South location, and the error $E(\vec{w})$ your altitude. This gives you a topological map of your error surface. We are looking for a location $\vec{w}$ in a valley.

Blind: In practice the weights $\vec{w} = \langle w_1, \ldots, w_m \rangle$ consist of tens or hundreds of thousands of real numbers making for a very high dimensional space to search. With 10,000 weights with 0 to 9 integer values, there are $10^{10,000}$ settings of the weights. A super computer on each atom of the universe working for the age of the universe couldn't dent this list!

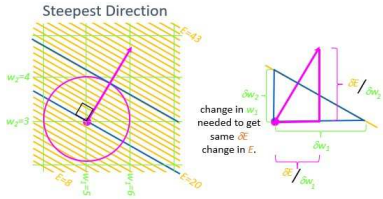
Linear Regression: If the machines are linear in the weights as in $M_{\vec{w}}(\vec{x}_d) = w_0 + w_1 x_{\langle d,1 \rangle} + w_2 x_{\langle d,2 \rangle} + w_3 \, x_{\langle d,2 \rangle}{}^2$, then there are formulas that will give you the weights $\vec{w}_{opt}$ that minimizes the error $E(\vec{w})$. Not so for complicated machines.

Gradient Decent: When I asked my young son how to find the top of the hill, he answered "Just keep going up." When ask which direction to head, he answered "in the direction that is steepest." When ask how do you know when you are at the top, he answered "When you can't go up anymore." This is how we will find the bottom of a valley. It might not find the *global* minimum, but we hope that it finds a machine that is good enough for our purposes. From this we get self driving cars!



We find a local optimal
not a global one.

Smooth/Differentiable: This error surface is such that an infinitesimally small change in any one of the weights w_k causes an infinitesimally small change in the output of the corresponding neuron of the neural net, in the output $M_{\vec{w}}(\vec{x}_d)$ of the machine, and in the error $E(\vec{w}) = \sum_d (y_d - M_{\vec{w}}(\vec{x}_d)^2$. Dividing the change in $E(\vec{w})$ by the change in w_k gives the derivative $\frac{\delta E}{\delta w_k}$. The vector of these derivatives $\Delta(\vec{w}) = \left\langle \frac{\delta E}{\delta w_1}, \frac{\delta E}{\delta w_2}, \ldots, \frac{\delta E}{\delta w_m} \right\rangle$ is called the *gradient*. This is computed efficiently by a process called *back-propagation* using the calculus learned in high school. The figure below provides a little of the math as to why this vector gives the direction of steepest decent. Intuitively, the weights w_k that influence correct answers are increased and those that influence wrong answers are decreased.

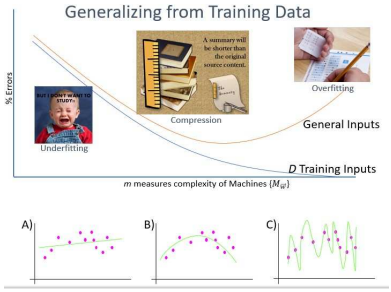


Algorithm: Method for finding weights $\vec{w}_{opt}$ that minimizes the error $E(\vec{w})$ is as follows:

Start with random $\vec{w} = \langle w_1, \ldots, w_m \rangle$
Repeat:
$\langle loop{-}invariant \rangle$: We know where we are $\vec{w} = \langle w_1, \ldots, w_m \rangle$.
Calculate our height $E(\vec{w})$.

Calculate the direction and slope of steepest descent.
Change $\vec{w}$ by a small step in this direction and distance proportional to the slope.
Stop when every direction is up.

Generalizing: This machine is optimized to do well on the training data. But how well does it generalize to other inputs that it has never seen before? There is a "no free lunch" theorem that says that no one method works in all cases. But there are a few principles. Consider the following three examples, A, B, & C. Each dot give the $\langle x_d, y_d \rangle$ of one training data item. The curve gives the answer that the machine gives for each input x. Ask yourself which of these three gives the best answers for values of x that it has never seen before?



Underfitting: A is an example of *Underfitting* in which the class of machines just doesn't have the capacity to learn the material.

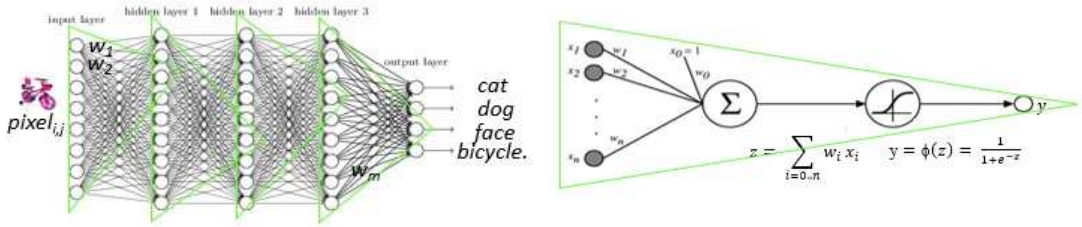
Overfitting: C is an example of *Overfitting* in which the class of machines is so powerful that is can memorize the answers for just about any function. The problem is that if you cheat on the exam by making a table of all the answers you were taught, then you learn nothing and are stuck when a new question comes.

Compression: B seems like a good balance of under and over fitting. On a test, if you summarize the learned material, compressing it to all fit on one "cheat sheet", then hopefully you *understand* it and do well when a new question comes. Compression occurs when the weights $\vec{w} = \langle w_1, \ldots, w_m \rangle$ with which the machine learns, does not contain enough bits of information to store the training data answers $\langle y_1, \ldots, y_D \rangle$.

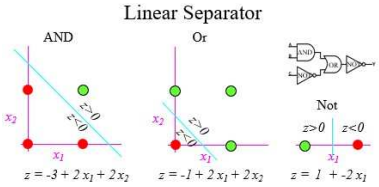
Regularization: A common way to find this balance is to minimize not just the error but a sum of the error and of some measure of how complex your machine is. William of Ockham says "All things being equal, the simplest solution tends to be the best one."

Theory: Learnable Probably Approximately Correct (PAC): If on the randomly chosen training data, a machine is found that gives good answers, even though compression is required, then we can prove that with high probability this same machine will also work well on never seen before instances. The proof does not measure the quality of the machine but of the training data. If the data is chosen randomly then with high probability it is *representative* of the entire universe of possible data.

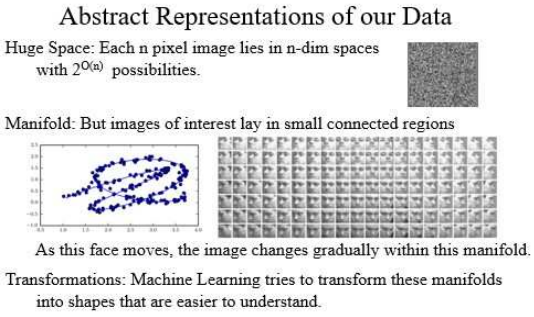
Neural Networks: Let's now understand neural networks in more detail. It is made of layers of artificial neurons. The input layer has an input wire for each pixel of the input image receiving a value between 0-1 indicating the pixel's brightness. The output layer has a wire for each category indicating the "probability" that the input image is in fact a cat or dog. In between, the layers are said to be *hidden*. These have no human designated meaning. Meaning is "learned" by choosing weights $\vec{w} = \langle w_1, \ldots, w_m \rangle$.



When a *neuron* in the brain fires, it sends a signal to its neighbors across its synapses. These synapses have different weights in their influence. When a neuron's incoming signal reaches a threshold it fires. Similarly, an artificial neuron takes a real valued signal x_i along each of its incoming edges, multiplies this by the edge's weight w_i, and fires if the weighted sum $z = \sum_{i=0..n} x_i w_i$ reaches its threshold. In this way, a neuron is able to compute a *And, Or,* or *Not* gate and hence is powerful enough to be able to compute anything a digital computer can compute. So we can later do gradient decent, we change the threshold activation $y = \sigma(z)$ so that it transitions gracefully. Instead of YES/NO we allow maybe so, i.e. if the weighted sum is close to its threshold, then it outputs a half instead of 0-1.



Vectors: Let's try to understand how to best think about all of these numbers. A data input $\vec{x} = \langle x_1, \ldots, x_m \rangle$, eg a image of a cat, is just a large tuple of real values. As such it can be thought as a point in some high dimensional vector space. Whether the image is of a cat or a dog partitions this vector space into regions. Classifying your image amounts to knowing which region the corresponding point is in. A *linear separator* separates two regions with a plane. It is unlikely that this will work to separate cats from dogs. Instead, each layer of the neural network transforms the input $\vec{x}$ until it can categorize it.



Correlation of Vectors: Machine Learning is all about finding complex correlations. Suppose I am starting a dating service. For each client I will collect properties. For each pair of clients, we measure how compatible they are. The first two clients in the figure below each like movies a lot. This adds $15 \times 20 = 300$ to their compatibility. They are both neutral on nature, adding only $3 \times 2 = 6$. They differ on being extroverted making them $(-20) \times 30 = -600$ incompatible. Summing over all properties gives a total compatibility of -874, i.e. incompatible. The first and third client with 910 are much more compatible.

Dot Product

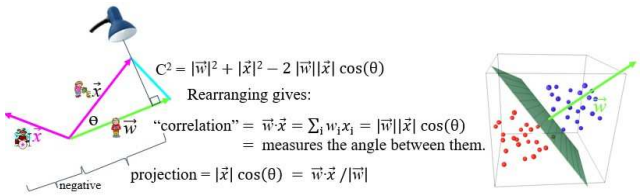
	Likes animals	Extrovert	Partier	Likes movies	Likes nature	Likes music
	-10	-20	-20	15	3	-9
	-9	30	20	20	2	30
	-8	-10	-15	10	30	-10
	(-10)×(-9) = +90	(-20)×30 = -600	(-20)×20 = -400	15×20 = 300	3×2 = 6	(-9)×30 = -270

Total Correlation: -874

	(-10)×(-8) = 80	(-20)×(-10) = 200	(-20)×(-15) = 300	15×10 = 150	3×30 = 90	(-9)×(-10) = 90

Total Correlation: 910

Denote the vector/tuple of values for the first client with $\vec{w} = \langle w_1, \ldots, w_m \rangle$ and with the second with $\vec{x} = \langle x_1, \ldots, x_m \rangle$, their compatibility$(\vec{w}, \vec{x})$ is computed as $\sum_i w_i x_i$. This is called the *dot product* $\vec{w} \cdot \vec{x}$ between these vectors. If these vectors are thought of arrows in high dimensional space then the angle between them is given by $cos(\Theta) = \frac{\vec{w} \cdot \vec{x}}{|\vec{w}| \cdot |\vec{x}|}$. If $\vec{w} \cdot \vec{x}$ is positive, the angle Θ is less than ninety degrees and if it is negative, then it is more. If you shine a light perpendicular down onto $\vec{w}$, the length of $\vec{x}$'s projection will be $|\vec{x}| \cdot cos(\Theta) = \frac{\vec{w} \cdot \vec{x}}{|\vec{w}|}$. For some given distance d, the set of all vectors $\vec{x}$ with this projection length equaling d are those within the hyper-plane perpendicular to $\vec{w}$ a distance d from the origin. When $\frac{\vec{w} \cdot \vec{x}}{|\vec{w}|}$ is less than this d, then $\vec{x}$ is on the near side of this plane and when more than it is on the far side.

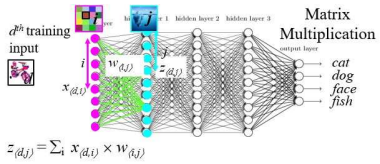


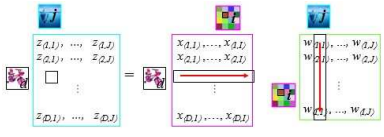
Linear Layer = Matrix Multiplication: The simplest layer of a neural network is linear. A novice reading a machine learning paper might not get that many of the symbols are not real numbers but are matrices. Hence the product of two such symbols is matrix multiplication.

Consider feeding the d^{th} training data input $\vec{x}_d = \langle x_{\langle d,1 \rangle}, \ldots, x_{\langle d,m \rangle} \rangle$ into the neural network. Its i^{th} value $x_{\langle d,i \rangle}$ is fed into the network's i^{th} input wire. The set of all of these values $x_{\langle d,i \rangle}$ can be organized into a matrix $X = \left[\, x_{\langle d,i \rangle} \, \right]_{\langle d,i \rangle}$ whose d^{th} row corresponds this d^{th} training data input $\vec{x}_d$.

Let $w_{\langle i,j \rangle}$ denote the weight of the edge from the i^{th} input wire to the j^{th} artificial neuron in the first layer of the network. These values can be also be organized into a matrix $W = \left[\, w_{\langle i,j \rangle} \, \right]_{\langle i,j \rangle}$.

The total incoming signal into the j^{th} hidden neuron on the d^{th} input $\vec{x}_d$ is denoted $z_{\langle d,j \rangle}$. Again these values form a matrix $Z = \left[\, z_{\langle d,j \rangle} \, \right]_{\langle d,j \rangle}$.

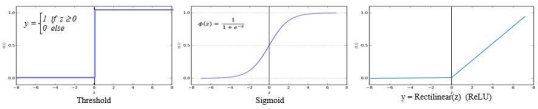




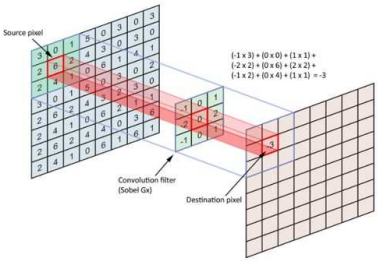
This j^{th} hidden neuron has many incoming edges, the i^{th} of which is labeled with weight $w_{\langle i,j \rangle}$ and receives the input value $x_{\langle d,i \rangle}$. These are multiplied and these products for the different incoming edges are added together, i.e. $z_{\langle d,j \rangle} = x_{\langle d,1 \rangle} \times w_{\langle 1,j \rangle} + \ldots + x_{\langle d,I \rangle} \times w_{\langle I,j \rangle} = \sum_i x_{\langle d,i \rangle} \times w_{\langle i,j \rangle}$. Note that this is the exact calculation computed in the matrix multiplication $Z = X \times W$ as the $\langle d, j \rangle$ entry of matrix Z is given by the dot product of the d^{th} row of matrix X and the j^{th} column of matrix W, summing over index i.

Non-Linear Layer: Given signal $z_{\langle d,j \rangle}$, the j^{th} hidden neuron ideally either fires or it doesn't. The picture is either a cat or not. One might model this by having the hidden neuron fire if this

signal $z_{\langle d,j\rangle}$ passes a threshold. Typically, a constant $w_{\langle 0,j\rangle}$ is added into the sum so that the threshold is shifted to zero. To be more graceful, a typical activation function used is the sigmoid function $y_{\langle d,j\rangle} = \sigma(z_{\langle d,j\rangle})$ that has the hidden neuron output close to one when input sum $z_{\langle d,j\rangle}$ is very positive, output close to zero when it is very negative, and in between it changes gracefully. The problem with the sigmoid activation function is that its derivative and hence the gradient of steepest decent is close zero for large activation values z making learning slow. To solve this problem, the rectilinear activation function does not change the signal, i.e. $y = z$ when it is positive and zeros it, i.e. $y = 0$, when negative. Note for positive z, the slope here is always one making learning faster. Recent work has shown that this activation function works surprisingly well for many applications.



Convolution and Recurrent Layer: Suppose the input is an image and the neural network is trained to find a cat. This problem is invariant over the location of the cat, namely the network should answer yes no matter where in the image the cat is located. This invariant can be built into the design of the neural network by having a set of weights be learned to find a feature in some small part of the image and to use those same weights translated all over the image. Similarly, if the input in a stream of speech, then the same set of weights can detect the word cat at the beginning and at the end of the stream. These changes decrease the number of weights that the neural network has, making learning easier, decreasing computation time, and decreasing overfitting.



The Singularity: If technology grows just slightly faster than exponentially, then at some point in time, the expected about of technology will be infinite. What will life look like after that?