Senior Project (draft)
Lafayette College
Department of Mathematics

Wah Loon Keng*

November 29, 2015

Abstract

We conduct a survey study on machine learning, its mathematical foundations, general techniques, and the latest progress in both the academia and the industry. However, most machine learners excel only at the task they are trained for - this is a huge limitation. Motivated by this, we study a new paradigm called the Never-Ending Language Learning (NELL)[1] and its potentials to overcome the singular nature of machine learners. Lastly, we design and implement a system based on NELL to predict the stock market performance. This is driven by the abundance of data, and the complex, interacting, non-singular nature of the task that a machine learner has yet to match a human on.

*Lafayette College, Easton, PA 18042, USA. kengw@lafayette.edu.

Contents

1 Introduction

We have just entered the golden age of machine learning. The field has in fact been around for decades, born from artificial intelligence and pattern recognition. It wasn't until recently that we had the level of computational power and the abundance of data to apply it.

The machine learners have accomplished some truly astonishing feats. They are systems that are trained with a copious amount of data, which allows them to surpass the performance level of human experts. Today, they are actively being researched and deployed by the biggest tech companies such as Google, IBM, Microsoft, Facebook. In fact, their major products are powered by machine learning - Google Adsense, IBM Watson, and Facebook's M - doing tasks such as ads suggestion, image recognition, medical diagnosis, and natural language processing.

Typically, machine learning is best suited for the tasks that cannot be solved precisely or efficiently by algorithms. Many of these are what humans excel at - image recognition, language processing, pattern deduction. These capabilities are so trivial to us, yet they are so difficult to mimic by traditional algorithmic approaches. The prime example of *"a child can easily recognize a cat in a picture, but a super computer can't"* best illustrates the failure of the algorithmic methods. Machine learning comes in to save the day.

In a nutshell, one can think of a it as a machine that "learns" from the data. On the implementation level, it is an automated regression software that can construct an impressively accurate model. The machine takes in a training set - input data where each entry is labeled with the intended output, then trains on it and learns to recognize the pattern. After that, when fed with new unlabeled input, it gives some predicted output based on its training. However, just like a typical regression model, its scope is strictly bounded to its training set. For example, a machine that recognizes cats cannot recognize a fish without extra training. Moreover, an image recognizer cannot diagnose diseases. In this sense a machine learner's scope is **singular**. We will address this issue later.

The mathematical foundations of machine learning is built atop regression theory and linear algebra. Next, we present the formalization of a problem in machine learning as well as the terminologies.

1.1 Formalization

The following formalization is standard in the literature, we reference Tom M. Mitchell's popular textbook *Machine Learning*[2].

Definition 1. *A computer program is said to* ***learn*** *from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*

We will use the terms *training* and *learning* interchangeably. Next, we define a **problem**.

Definition 2. *A **problem** is a triple $\{T, P, E\}$. The goal of a machine learner solving the problem is to output a learned target function $\hat{V}$ that approximates an ideal target function V. The target function maps from the vector of input features b to the set of output O.*

The experience E represents the input data. It is the set of points specifying the configurations of the instances of the problem. To use it for training, the data must be *labeled / classified* - every input data must have its output. This is needed for the performance measure P. In the context, it is understood that the data is always labeled.

Often the data set is split into three parts - the training set, validation set, and test set, for obvious statistical reasons. The first is used to weight-tuning (see below) that may yield multiple candidates, the second for validating and choosing the best candidate, and the last for testing against new input data and detecting overfitters.

The feature vector b, output O, and the function representation for $\hat{V}$ are defined by the programmer as deemed suitable. In a simple regression model, b is a list of chosen input features, O is the set of predicted values. Typically the target function $\hat{V}$ will include a set of weights $W = \{w_i \mid i \in \{1, 2, \cdots, n\}\}$ which are tuned during the learning. They serve as the "memory" of the machine to compute predicted output once it is trained.

Upon having an explicit representation, and given the training data, we can devise the learning mechanism. First, define an error term $\mathcal{E}$ between the training data and the actual machine output. Then, use the error term to devise an algorithm that tunes the weights iteratively to minimize the error. The algorithm terminates on meeting a minimum threshold error, that is when the machine can perform sufficiently good.

Example 1. *We provide an explicit example with the checkers game:*

The checkers learning problem:

1. *Task T: playing checkers*
2. *Performance measure P: percent of games won*
3. *Training experience E: games played against another computer*

Next we need to determine the representations of the available knowledge. For the input feature vector b, we can choose a few board states, for instance,

$$b = \langle x_1, x_2, x_3, x_4 \rangle$$

where the x_i's are respectively the numbers of black pieces, black kings, red pieces and red kings on the board. We can choose a simple linear combination as our target function,

$$\hat{V}(b) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4$$

where w_i's are the weights to be tuned during the training. Our output set O is the list of moves at each game step given the features x_i's at the time. We can define a bijection between the output range of $\hat{V}(b)$ and the O, but we will exclude it here.

Once we have a representation, devise a learning mechanism for training. Suppose now we have a set of positive (games won) training examples $\{\langle b, V_{train}(b)\rangle\}$. Define the error term using simple sum of squares

$$\mathcal{E} = \sum_{\langle b, V_{train}(b)\rangle \in \text{ training examples}} (V_{train}(b) - \hat{V}_{train}(b))^2$$

Then, using this, devise an algorithm A for training / weight-tuning. The error term above, when differentiated, yields gradient term of $\sim (V_{train}(b) - \hat{V}_{train}(b))$. One can imagine training as nodging a point closer to its true value, and the gradient gives the direction to nodge towards. This gives us a simple LMS algorithm for minimizing $\mathcal{E}$ using that gradient term:

Algorithm **LMS weight-tuning (learning) algorithm**

Initialize the weights w_i's to random values. For each training example $\langle b, V_{train}(b)\rangle$:

1. *Use the current weights to compute $\hat{V}(b)$*
2. *For each weight w_i, update it by*

$$w_i \leftarrow w_i + \eta \ (V_{train}(b) - \hat{V}_{train}(b)) \ x_i$$

where η is the parameter that moderates the stepsize. The algorithm terminates when w_i's stop changing, i.e. when the error becomes sufficiently small (zero).

One can quickly see that the example machine learner above is just a simple regressor. Obviously this toy model will not be a good checker player. Nevertheless, given a huge training data, if we choose the right knowledge representation, feature vector b, target function $\hat{V}$, error term $\mathcal{E}$, and the learning algorithm, the resultant machine can perform very well.

To suit a different type of problem, we can employ a different **representation** - which we define to be the set $\{T, P, E, b, O\}$, as illustrated in Table 1.

Moreover, we can employ different **learning mechanisms** - which we define as the set of target function and learning algorithm, $\{\hat{V}, A\}$. This gives rise to the different techniques, or types, of machine learners such as the Neural Nets, Support Vector Machines, Deep Learning and etc. We will investigate the popular types in the next section.

A huge variety of machine learners today are being used for different purposes - most of the leading tech companies tweak and design their own. They may have very different

T	P	E	b	O
image recognition	correct labeling	labeled images	array of pixels	label of objects
driving a car	successful trip	test drives	traffic condition	wheel, pedals
text translation	correct translations	pretranslated texts	semantics	translated text
ads suggestion	ads click rate	ads metadata	user profile	ads displayed

Table 1: Different representations for various problems.

representations and **learning mechanisms**, but the overall idea is still the same - gather labeled data, define the representations and learning algorithms, and train. The entire process is very modular, and the resultant machines are **stagnant** - the machine no longer improve once training is done; and **singular** - they can only function very narrowly only on the tasks they are trained for. For our discussions, we call this the **stagnant & singular paradigm**. In fact, this paradigm can readily be observed across most of the popular machine learners today. This will serve as the primary motivation of our work.

Before getting to the discussion on the paradigm, we take a quick look at the most common and powerful machine learners. These are derived from the same **stagnant & singular paradigm** by varying the **representations** and **learning mechanisms**.

1.2 Decision Tree

Decision tree is a popular method stemmed from artificial intelligence, used for hypothesis-searching and inductive reference. It uses a *tree representation*, where each node is a feature variable, and each edge is a value the feature takes on. The leaf nodes are the classification (output) values. Every path from the root to the leaf represents an instance of the feature b and its output o. Moreover, different paths can be combined with logical OR and AND to yield a more general hypothesis space.

For the *learning mechanism*, a common method is to use the entropy measure and entropy gain while branching on each node (making a decision) while descending the tree. Suppose the data has c classifications (e.g. $c = 2$ for binary classifiers), the entropy is defined as

$$Entropy(D) = \sum_{i=1}^{c} -p_i \ log_2 \ p_i$$

where D is the data, p_i is the proportion of samples of class i in D. This allows us to judge the performance P and search for the target function $\hat{V}$ by using the entropy gain for each feature b,

$$Gain(D, b) = Entropy(D) - \sum_{v \in Values(b)} \frac{|D_v|}{D} Entropy(D_v)$$

where D_v is the subset of D whose feature vector assumes the values equals to v.

Each path in the tree is a candidate hypothesis (collection of contributing features) to consider. Given the multitude of paths in the tree, the algorithm will end up with a set valid and consistent hypotheses. To resolve this, we simply use the **Occam's razor** - choose the simplest hypothesis that fits the data. This will also prevent the initial overfitting.

1.3 Artificial Neural Network (ANN)

Often called just the "neural net", ANN was inspired by the biological system of neurons in the brain (note that it is inconsistent with the biological version). This is one of the earliest techniques, and has improved over the decades; the first practical usage was in the 80's, and today it is one of the most popular with countless variations. Here, we will describe the original and the most common designs.

Neural network is a practical method for learning examples that are real-valued, discretized, and multi-dimensional. It is so general that it has been applied successfully to image recognition, pattern-identification, robotics, speech and many more. Its popularity is primarily due to its power to learn accurately and its simplicity.

An ANN is a directed graph whose nodes are the feature variables, and edges are the weights. It is ordered into layers of nodes that are adjacently bipartite, i.e. every node in each layer connects (directionally) to all the nodes in the next layer. The first is called the *input layer*, the last the *output layer*, and the rest in between the *hidden layers*. A node is also called a *unit*, and those in the *hidden layers* are called *hidden units*.

The *input layer* corresponds to the feature vector b - each unit takes the value for each feature variable. Similarly the output layer corresponds to the output o. Each iteration of training involves a series of computation from the input, through the hidden layers, to the output. Across the net, the outputs become the inputs for the next layer.

Suppose we have at layer i indexed nodes $x_{i,1}, x_{i,2}, \cdots, x_{i,p}$ bipartite to nodes $x_{i+1,1}, x_{i+1,2}, \cdots, x_{i+1,q}$ at layer $i+1$. We can simply represent as the edges with their weights with subscript indicating the connection. For example, the first node $x_{i,1}$ connects to all the edges in the next layer via $w_{i,1,1}, w_{i,1,2}, \cdots, w_{i,1,q}$.

Then, we can define the function for mapping input values at $x_{i,1}, x_{i,2}, \cdots, x_{i,p}$ to an output value at $x_{i+1,j}$, i.e. we can define a target function for an output node $x_{i+1,j}$

$$\hat{V}_{i+1,j} : \{w_{i,1,j}, w_{i,2,j}, \cdots, w_{i,p,j}\} \times \{x_{i,1}, x_{i,2}, \cdots, x_{i,p}\} \mapsto x_{i+1,j}$$

Typically, an inner product is sufficient to yield a good ANN. For this we have the form

$$\hat{V}_{i+1,j} = \langle w_{i,1,j}, \cdots, w_{i,p,j} \rangle \cdot \langle x_{i,1}, \cdots, x_{i,p} \rangle = w_{i,1,j} x_{i,1} + \cdots + w_{i,p,j} x_{i,p}$$

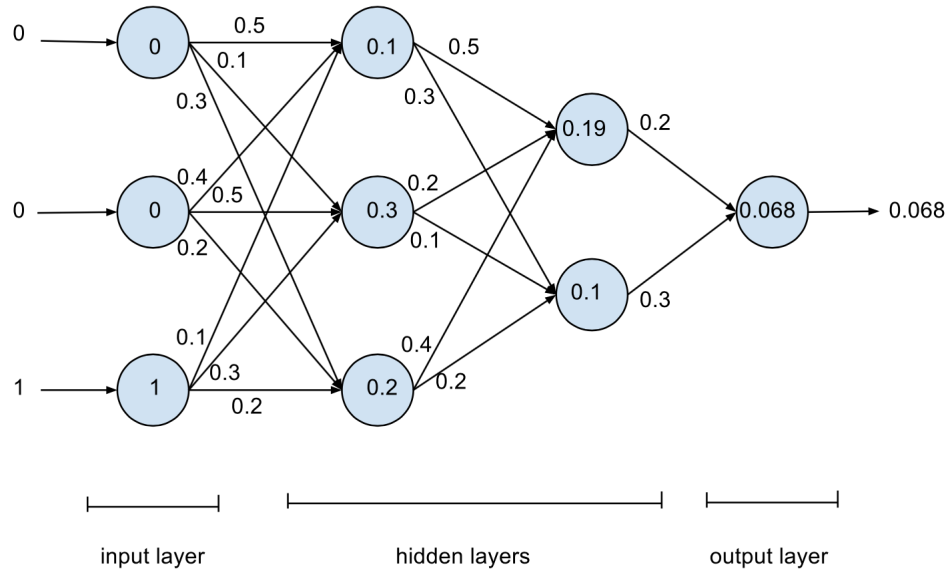

Figure 1: An ANN with 3 input units, 2 hidden layers each with 3 and 2 hidden units, and 1 output unit. The target function is the dot product between the incident values from the previous layer and their edge weights.

An explicit example is provided in Figure 1. The feature vector input is $b = \langle 0, 0, 1 \rangle$; the value

$$x_{2,1} = \hat{V}_{2,1} = w_{1,1,1}x_{1,1} + w_{1,2,1}x_{1,2} + w_{1,3,1}x_{1,3} = 0.5 \cdot 0 + 0.4 \cdot 0 + 0.1 \cdot 1 = 0.1$$

If we wish, we can define the target function $\hat{V} : b \mapsto o$ for the entire ANN by taking the ordered composition of the sub-target functions defined above, but this is cumbersome. Often the graph representation is used.

We can create variations of ANN by tweaking the target function $\hat{V}$, the error term $\mathcal{E}$, and the training algorithm A. Moreover, we can change the graph structures by adding more hierarchies (convolutional, deep learning) or *backedges* to create cycles (recurrent neural network RNN).

With data $\{\langle b, V(b) \rangle\}$, the training goes as usual. A single iteration of training starts from the input layer and ends at the output layer. The error term is then used with the tuning algorithm A to tune the edge weights, i.e. use the errors to update the weights. Reiterate the process until the output error is smaller than a threshold. Unsurprisingly, training a large neural net can take up to months, but the results can be very impressive. Below, we describe some of the most powerful versions that have recently made the news.

1.4 ANN with Perceptrons

Given the general design above, we can simplify our notations to aid discussions. Focusing on a subnetwork, look at a single output unit o with incoming inputs $x_1, x_2, \cdots, x_k$ and weights $w_1, w_2, \cdots, w_k$, the target function for a perceptron unit is

$$o(x_1, \cdots, x_k) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_k x_k > 0 \\ -1 & \text{otherwise} \end{cases}$$

Concisely, we write

$$o(\vec{x}) = sign(\vec{w} \cdot \vec{x})$$

The training rule for the algorithm is a simple update,

$$w_i \leftarrow w_i + \eta(t - o)x_i$$

where η is the stepsize moderation called the *learning rate*, t is the target output value, o the actual perceptron output, x_i the input node value.

1.5 ANN with Gradient Descent

The gradient descent can be seen as a refinement to the perceptron. Its target function is

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

The unit is called a *linear unit* for an obvious reason. The error term is defined as

$$\mathcal{E}(\vec{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

where D is the data, t_d is the target output value, o_d the unit output for a data point d. This will guide the weight-tuning rule for the algorithm, which also gives its name.

Given the formalism, we can equivalently view as a vector space. Our task then is to find the set of weights that yields the right target function. Minimizing the error term is then equivalent to minimizing the error surface in the vector space. We know this is done by stepping in the direction of the steepest descent along the error surface, which is simply the gradient of the error term

$$\nabla \mathcal{E}(\vec{w}) = \langle \frac{\partial \mathcal{E}}{\partial w_0}, \frac{\partial \mathcal{E}}{\partial w_1}, \cdots, \frac{\partial \mathcal{E}}{\partial w_k} \rangle$$

Then, the tuning rule is simply

$$\vec{w} \leftarrow \vec{w} - \eta \nabla \mathcal{E}(\vec{w})$$

where η is the learning rate. When written in component form we get

$$w_i \leftarrow w_i - \Delta w_i = w_i - \eta \frac{\partial \mathcal{E}}{\partial w_i}$$

Algorithm **Gradient-Descent(Data, η)**

Each training example $d \in Data$ is of the form $d = \langle \vec{x}, t \rangle$, where $\vec{x}$ is the input vector, t the target output value, η the learning rate. Note that the partials are found using discretized iterations at step 2.2.2.

1. Initialize the weights w_i's to small random values.
2. Until termination condition (defined separately), do
   - 2.1. Initialize each Δw_i to 0
   - 2.2. For each $\langle \vec{x}, t \rangle \in Data$, do
     - 2.2.1. Input $\vec{x}$ and computer output o
     - 2.2.2. For each unit weight w_i, do

$$\Delta w_i \leftarrow w_i + \eta(t - o)x_i$$

   - 2.3. For each linear unit weight w_i, do

$$w_i \leftarrow w_i + \Delta w_i$$

1.6 ANN with Backpropagation

The invention of *Backpropagation* by Geoffrey Hinton and Yann LeCun reinvigorated ANN after its loss of popularity due to practical limitations. First, we improve the target function by using a sigmoid (logistic) function, yielding a *sigmoid unit*

$$o = \sigma(\vec{w} \cdot \vec{x})$$

where

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

It has some very nice properties. The function maps the entire real domain into $(0, 1)$ like a "squeezing function" with the outlier points getting pushed to the narrow ends of the interval; it increases monotonically; it is smooth with a convenient derivative $\sigma'(y) = \sigma(y)(1 - \sigma(y))$, which is computationally cheap.

Next, *Backpropagation* can efficiently train a large multilayer network. We can directly generalize the error term as the sum over all nodes at a layer,

$$\mathcal{E}(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{d,k} - o_{d,k})^2$$

The following algorithm constructs an ANN that uses *Backpropagation* with the number of units as parameters.

Algorithm **Backpropagation(Data, η, n_{in}, n_{out}, n_{hidden})**

Each training example $d \in Data$ is of the form $d = \langle \vec{x}, t \rangle$, where $\vec{x}$ is the input vector, t the target output value, η the learning rate. n_{in}, n_{out}, n_{hidden} are respectively the numbers of units in the input, hidden and output layers. Double indices are ordered as *from, to.* Note that the partials are found using discretized iterations at step 3.1.2.

1. Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
2. Initialize all weights to small random values.
3. Until termination condition (defined separately), do
   - 3.1. For each $\langle \vec{x}, t \rangle \in Data$, do:
     - 3.1.1. (Propagate the input through the network) Input $\vec{x}$ and computer output o for every unit
     - 3.1.2. (Propagate the errors backward through the network) For each output unit o_k, compute its error term δ_k by

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

     - 3.1.3. For each hidden unit h, compute its error term δ_h by

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{hk} \delta_k$$

     - 3.1.4. Update each weight w_{ij} by

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

       where

$$\Delta w_{ij} = \eta \ \delta_j \ x_{ij}$$

There are obvious caveats to *Backpropagation*, namely convergence and local minima. The former can be resolved by added heuristics in the externally-defined termination condition;

the latter can be overcome by updating the algorithm with *stochastic gradient descent* instead. Fortunately for most practical purposes, these limitations can safely be ignored.

This ANN is in fact so powerful that it can represent every Boolean function, approximate any continuous functions, as well as arbitrary functions. The *Universal Approximation Theorem*[3] states that a feedforward ANN with three layers of units can achieve this generality.

Since its invention in the 80's it has become the primary method for ANN. Indeed thus far in this paper, *Backpropagation* is the most powerful machine learner, until the recent rise of the more powerful *Deep Learning.*

1.7 Deep Neural Network (DNN)

The most popular and powerful machine learning technique today is undoubtedly deep learning, given the impressive accomplishments by *IBM Watson, Facebook M*, and *Google Tensorflow* this year. The theoretical foundation has been around for decades, but it wasn't until now that we had the level of computational power and abundance of data to apply it.

Deep learning is implemented as a Deep Neural Network (DNN), which is itself a neural net. The term *deep* stands for the depth of the hidden layers of DNN. Whereas most ordinary ANNs are "shallow" with less than 3 hidden layers, DNNs are "deeper" with many (usually over 3) hidden layers. It turns out that having more hidden layers can make a neural net immensely more powerful that it can represent a larger class of functions.

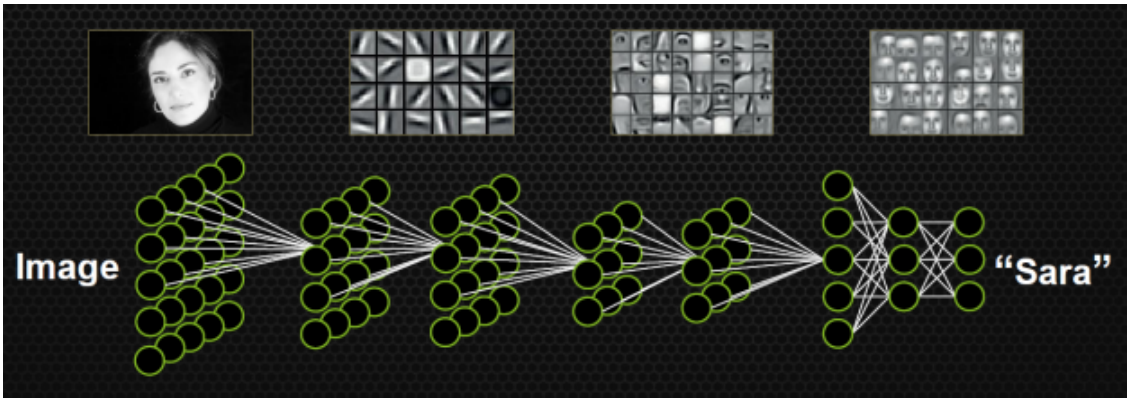

Figure 2: DNN that mimics the hierarchical structure of the brain. Going from left (input) to right, features get generalized from the specifics into more generic and abstract representations. At the top level (rightmost), `Image from NVIDIA.`

Having more hidden layers allows DNNs to progressively generalize features and abstractions. Going from the first to the last hidden layer, a DNN can capture features that are decreasingly specific and increasingly generic. For instance in face recognition, a DNN may capture the specific shades, then the eyes, nose, mouth features, then the shape, and finally a more

invariant representation of the face of a person, say Sara in Figure 2. It is the top level generalization that allows it to recognize a person's face regardless of the local variations, such as the head's orientation, the lighting, and the facial expression. In this aspect, DNNs are more "brain-like" than other machine learners, as we shall discuss more in depth later.

Whereas ANNs are *supervised* (requires the training data to be labeled), There are two types of learning for DNNs, both of which are not supervised. *Semi-supervised learning* is where unlabeled training data is drawn from the same prior distribution as the labeled training data; *self-taught learning* does not have that requirement.

Next, we look at an *autoencoder* - a basic but fundamental component of a DNN.

1.7.1 Autoencoder

The autoencoder is a neural net that learns the structure from an unlabeled data set. It has one hidden layer; the non-output layers (the first two) each has an extra *bias* neuron that accepts no input and constantly outputs 1. Given an unlabeled data set of n points, subscripted with u, $\{x_u^{(1)}, x_u^{(2)}, \cdots, x_u^{(n)}\}$, the autoencoder sets the outputs $y^{(i)} = x_u^{(i)}$, and train on the data $\{(x_u^{(1)}, y^{(1)}), \cdots, (x_u^{(n)}, y^{(n)})\}$ as usual. Since the outputs are set to the input, we can view the autoencoder as a neural net that learns the identity function, and its tuned weights would uncover the structure in the data.

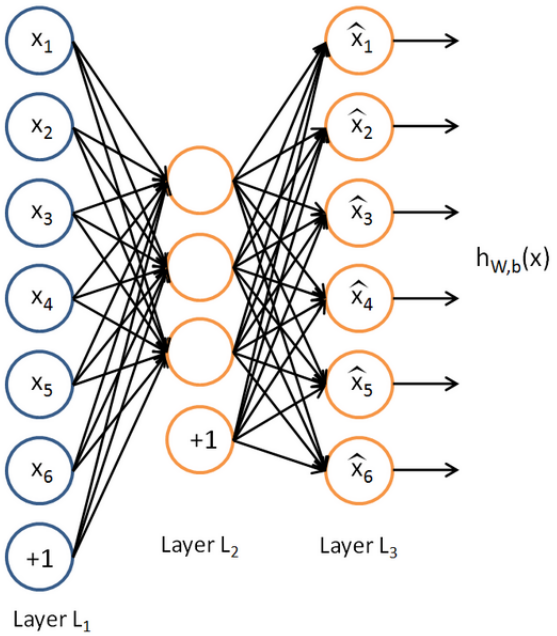

Figure 3: An autoencoder. The units with +1 are the bias units. The output units $\hat{x}_i$ will try to converge to x_i, as autoencoder mimics an identity function. `Image from UFLDL.`

We now introduce the tensorial notation used for DNNs.

Symbol	*Meaning*
x	The input features, $x \in \mathcal{R}^n$.
y	The target values (vector), $y = x$ for autoencoder.
$(x^{(i)}, y^{(i)})$	The i^{th} training example. For autoencoders, $y^{(i)} = x^{(i)}$.
L_l	The l^{th} layer of the net.
$b_i^{(l)}$	The bias unit at layer l that points to unit i in layer $l+1$ (think of it as the edge); outputs 1.
$W_{ji}^{(l)}$	The weight going out from the i^{th} node in layer k to the j^{th} node in layer $k+1$. Note the ordering of the indices, which is different from the previous sections.
$z^{l+1} = W^{(l)}a^{(l)} + b^{(l)}$	The outputs of all unit from layer l, in compact tensorial form. Note that $a^{(1)} = x$.
$f(\cdot)$	The activation function that transforms the input z at each node into the activation $a = f(z)$. Applies element-wise to a tensor. Typically, we use a sigmoid function $f(z) = 1/(1 + e^{-z})$.
$a^{(l)}$	Activation, i.e. the transformed outputs of all units at layer. $a^{(l)} = f(z^{(l)})$.
$h_{W,b}(x) = a^{(3)} = f(z^{(3)})$	The hypothesis of an autoencoder, given input features x, parametrized by the edge weights in the tensors W, b.

Table 2: Tensorial notation for DNNs.

1.7.2 Hybrid Method

One of the biggest hurdles to machine learing is the scarcity of labeled data. Every data point has to be manually labeled by humans - not every picture of a cat on the web is already tagged with "cat". Therefore, despite having a large trove of data today, most of it is unlabeled and thus remains unusable by supervised machine learners.

The problem necessitates new approaches to make use of the unlabeled data - self-taught learning and semi-supervised learning as mentioned in the section on Autoencoders. However, a machine trained solely by these methods only learns the structure of the data, and cannot be very useful as its output approximates its input - it cannot be used as a general predictor.

Fortunately, we can do better by utilizing both the unlabeled and labeled data sets via a hybrid approach. First, we train the autoencoder to "memorize" the structure of the unlabeled data - this is called *pretraining*. Then, we replace its output layer with a supervised ANN, and train the new network on the labeled data - this is called *fine-tuning*. The result can have a far greater representational power. This is the basis of a DNN, and is illustrated in Figure 4, 5 and 6.

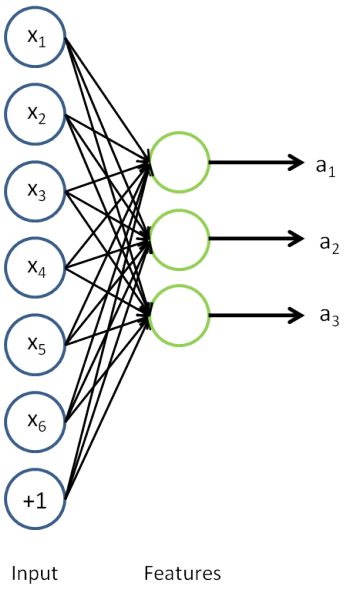

Figure 4: Self-taught learning: Pretraining Sparse Autoencoder - simply remove the third layer from the Autoencoder after training. `Image from UFLDL.`

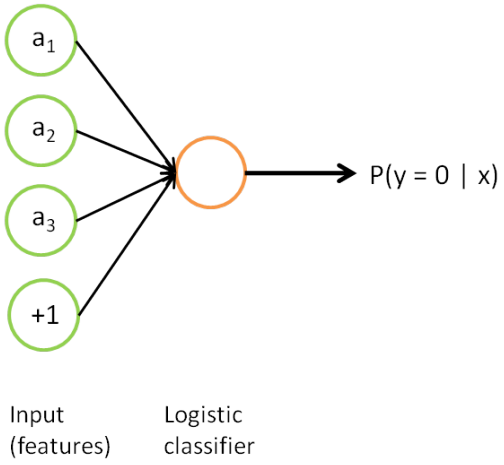

Figure 5: Supervised learning: Training Logistic Classifier (an ANN) using the activation units from the Sparse Autoencoder plus a bias unit. `Image from UFLDL.`

The intuition is that *pretraining* "coarse-tunes" the hidden layers into a guided configuration (of edge weights), which gives a better headstart than random weight initialization for training (*fine-tuning*) on the labeled data set. As mentioned, this requires a large set of unlabeled training data. On top of that, we must have a comparably large set of labeled training data to use *fine-tuning*. This hybrid method will significantly improve the performance of the network, and is key to the tremendous power of DNNs.

1.7.3 Stacked Autoencoders

Before we present a sample construction for a deep network with more hidden layers, two design features are crucial to making DNNs powerful and practical:

1. The activation function $f(\cdot)$ must be non-linear. Otherwise, when composing the functions over the layers, the linear composition of linear functions is also linear, and thus would result is a far more restrictive representational power. Commonly the non-linear activation function of choice is the sigmoid function.

2. "Compactness" can be achieved in the sense that k hidden layers can represent what $k-1$ layers cannot unless the latter has exponentially more hidden units. Whereas to increase the representational power by adding a hidden layer, the number of units only grows polynomially. The deeper the network, the more powerful it is.

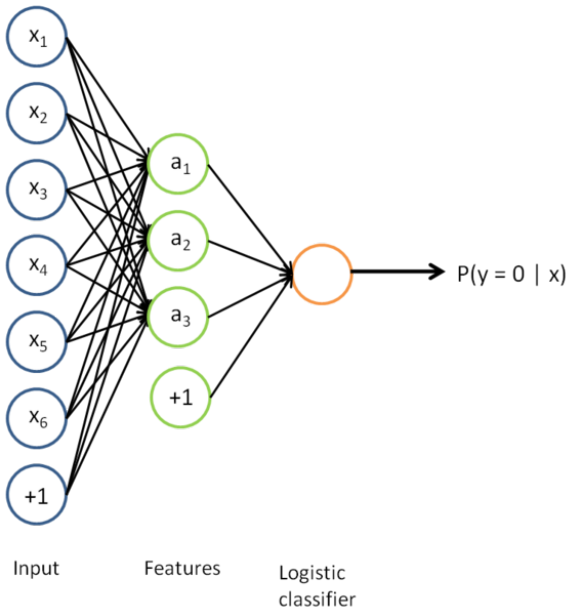

Figure 6: Semi-supervised learning as the basis for deep-learning: The network obtained by combining the Sparse Autoencoder from pretraining and the logistic classifier from supervised training. `Image from UFLDL.`

Having more hidden layers also pose the following problems.

1. *Diffusion* occurs, in which the error adjustment during backpropagation with gradient descent diminishes across layers. This makes it harder for the front layers closer to the input to converge.
2. A much larger amount of data is needed for training, and in some applications this is unfeasible.
3. The functions approximated by the deeper network is highly non-convex, and therefore may converge to some bad local optima.

The solutions to these are

1. Greedy layer-wise training: train the network with l layers, then add another hidden layer and retrain the network with $l + 1$ layers.
2. Utilize the abundant unlabeled data via *pretraining*.
3. After pretraining, the starting point is better, and *fine-tuning* will often lead to a good local optima in practice.

Generalizing from the previous construction, we now look at the *Stacked Autoencoders*. The extension is straightforward. Below, we use the greedy layer-wise training on the autoencoder - train the network with l layers, then stack another autoencoder on top and retrain the network with $l+1$ layers, therefore the name *stacked autoencoder*.

Formally, a *stacked autoencoder* is network of multiple layers of autoencoders. Each autoencoder (of 3 layers in total) at layer l is trained on unlabeled data, then stripped of its output layer. The output from its hidden layer is then used as the input layer of the next added autoencoder at layer $l+1$ (thus adding a hidden and an output layer with the proper bias units). This process repeats up until we have the desired number of hidden layers. The training at layer l for the autoencoder can be done locally while freezing the parameters of the other layers, which the input is injected and propagated to it from the first input layer of the network.

The activation units at layer l is written in the tensorial notation as

$$a^{(l)} = f(z^{(l)})$$

$$z^{(l+1)} = W^{(l)}a^{(l)} + b^{(l)}$$

Finally, when the greedy layer-wise pretraining using the unlabeled data is complete, a classifier (logistic, softmax, etc.) is added to the end of the network. We can now perform *fine-tuning* supervised training using the labeled data. The weights of the other layers shall now be allowed to change, as the entire network is being tuned. Backpropagation with gradient descent can be a good method for fine-tuning.

We give explicit illustrations of a deep network, which is a stacked autoencoder with 2 hidden layers, in Figure 7, 8, 9, 10 that are easier to grasp visually.

autoencoders n tensorial notations

Hawkins CLA Andrew Ng, One learning algorithm

performance, 3 layers few units suff to approx most functions?

Each round of training takes the input

Input layers X output layers recursive propagation T, P, E, b, O V, A

1.8 Applications

Google translate google ad sense google deep dream, deep mind Google deep learning (play game!) IBM watson

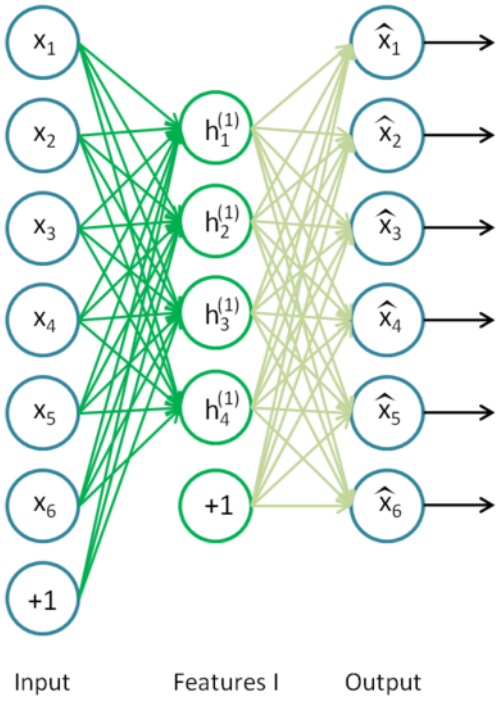

Figure 7: The first autoencoder, pretraining on the unlabeled data set with 6 input dimensions, and 4 hidden units. The pretraining will tune the edge weights to capture the first-order structure of the data. The output layer will then be discarded for adding the next autoencoder. `Image from UFLDL.`

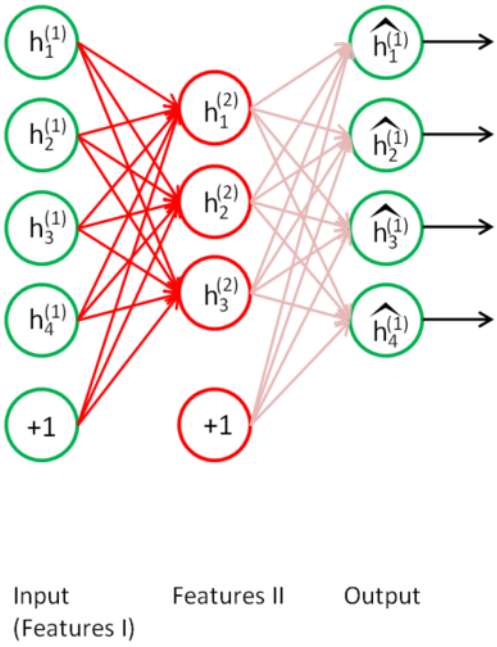

Figure 8: Supervised learning: Training Logistic Classifier (an ANN) using the activation units from the Sparse Autoencoder plus a bias unit. `Image from UFLDL.`

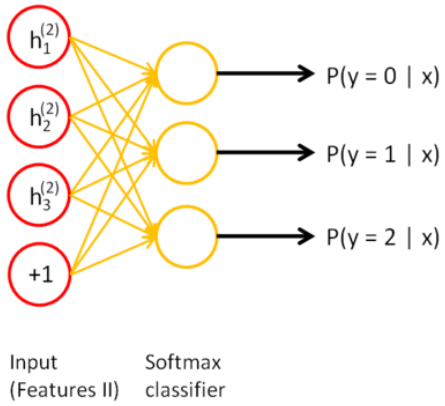

Figure 9: Self-taught learning: Pretraining Sparse Autoencoder - simply remove the third layer from the Autoencoder after training. `Image from UFLDL.`

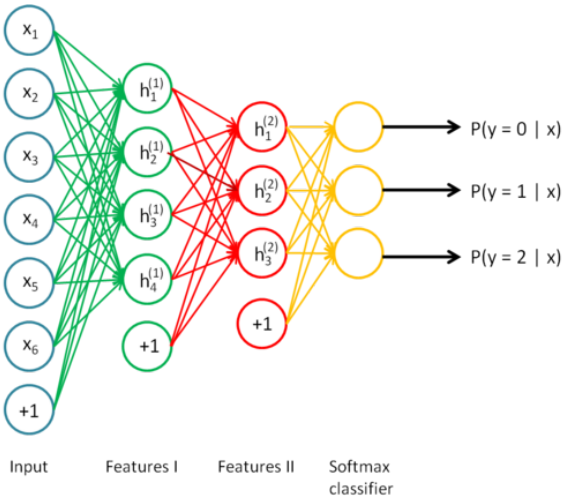

Figure 10: Supervised learning: Training Logistic Classifier (an ANN) using the activation units from the Sparse Autoencoder plus a bias unit. `Image from UFLDL.`

1.9 Paradigm: Powers & Limitations

2 NELL as a New Paradigm

2.1 Inspiration & Features

2.2 Architecture

2.3 Applications

2.4 Powers & Limitations

3 NELL for Stock Market Prediction

regardless of characteristics, in nearly every case we see that more wins are generated from the AI if they play first compared to when they play second. For some more closely related AIs, the odds of winning changes in its favor given the player order effect. From this we can conclude that for certain comparable AIs, player order may greatly affect the outcome of the game.

Furthermore, in every case, the trend line from the total army plot simply shifted its intercept value but held consistent slope. This intercept can be credited to first round play. After the first round (when the first point is calculated) the army advantage can be a one player advantage. This first round however rarely changes the results of the game. Consider figure 1.

Figure 1.

Here the green and red dots represent AI0s and AI4s percent total armies respectively over time. Again, the dots only represent games in which a definitive winner was found. Here we see that AI0 has a clear advantage but when the player order is changed, AI4 wins more games. Notice, however, while the trend line shifts down, that the slope stays constant. This implies that while the game is certainly affected by player order, on average, the game develops the same regardless of player order.

For the same AI match, the fun function provides two survival charts. Now consider figure 2.

Figure 2.

In the first plot we see that player two holds several armies until near round 50. Suddenly over the next two rounds however, we see an incredible drop off. AI0 has mostly defensive characteristics. From this it appears that AI0 condenses its portion of the graph and builds armies. Somewhere near halfway through the game, it branches out and gains several territories in a very small period of time. When the order of the game is reversed we see a much more even trend. Both AIs seem to lose armies at a consistent rate. This effect could be accredited to player order effect or the characteristics of the players in regards to their section of the sub graph.

While these trends, in general, hold true, one AI in particular defied the general trends. We will call this AI, AI3. AI3, in general, tended to produce more wins when playing in the second player position. This trend was tested more rigorously against 15 new AIs. From this data, the fun function produced the win records for both AI3 and the other AIs for both player order. The difference between wins was then calculated. That is $\Delta win = AI3\_wins - AIOpponent\_wins$. This figure was calculated for both player orders. This data is represented in figure 3.

Figure 3.

As in figure 3, on average AI3 wins 3.625 games while playing second in comparison to playing first where the AI won only 0.5 games on average. This in turn would seem to offer a solution to the player order effect, mentioned above. By simply playing the AI3 characteristics (Survival, Defensive, Cautious, Carry), on average the advantage would go to the second player, thus making player order obsolete.

Another nuanced observation shows that this conclusion may not be entirely true. Under further observation, another AI which we will call AI4 (Survival, Aggressive, Tactical, Rusher) seemed to be superior to AI3 in both player orders. A more rigorous study of these two AIs revealed that the assumption was correct. The fun function showed that on average, AI4 wins more often than AI3. Consider figure 4 and figure 5.

Figure 4.
Figure 5.

Notice that these two figures show very low correlation when player order changes. When AI3 goes first, it grosses more wins than AI4. When AI3 goes second, however, AI4 wins the majority of the games. While by a narrow margin, when AI4 goes first, AI4 wins 56 more games than AI3 in a 500 game trial. Thus if AI3 plays second they will on average lose to AI4.

4.2 AI behaviors and Surprises

In this project several AIs were posed against each other in a game of Risk. Initially, it was predicted that a large player order effect would affect the game. This hypothesis held true for the majority of AIs that were tested in this experiment. AI3 however, proved that by using the correct strategy, a player would actually hold an advantage by going second in the game. From this however, AI4 proved that their still exist AIs that can beat AI3 while being the first player to move.

From this project, we advise future Risk players to abstain from declaring their player strategy until the player order is determined. From this, we advise that second players play a cautious, defensive game, while first players play with aggressive tendencies. A consistent analysis of strategy will reveal the best strategy for either player to take throughout the duration of the game.

5 Conclusion

We have reformalized the board game *Risk* as a graph optimization, and implemented an AI to solve the problem, i.e. to play the game. Several decision algorithms were devised based on graph properties, and they in turn form the personalities of the AI. Our study of 4 binary traits, or a total of 16 personality variations of the AI, is merely the beginning. We have discovered some interesting AI behaviours and performance at a level beyond any human players. One can potentially investigate even more variations of the traits.

For future studies, an analysis of subgraphs would provide deep insight into the value of individual territories. The AIs are built in such a way that the priority function is based off of current demographic of troops and shape of their graph. Evaluating the graph as a whole would provide insight to the value of different subgraphs that would proactively affect the priority function. This conditional thinking is much more like the ideas of a human player and thus could inform future strategies.

Furthermore, game shifts could be analyzed using the fun function and the aforementioned subgraph analysis. This would allow for defensive players an understanding of which territories that are worth defending and worth conceding. This shift parameter would allow for insight to game length, army distribution, and other key factors.

6 References

1. Carnegie Mellon University, *NELL: Never-Ending Language Learning.* `http://rtw.ml.cmu.edu/rtw/`

2. Mitchell, Tom M, *Machine Learning.* New York: McGraw-Hill, 1997.

3. Cybenko., G. *Approximations by superpositions of sigmoidal functions*, Mathematics of Control, Signals, and Systems, 2 (4), 303-314. 1989.

4. Hawkins, Jeff and Blakeslee, Sandra. *On Intelligence.* New York: Henry Holt and, 2005.

5. Ng, Andrew et. al. *UFLDL Tutorial.* `http://deeplearning.stanford.edu/wiki/index.php/UFLDL_Tutorial`