Senior Project (draft)
Lafayette College
Department of Mathematics

Wah Loon Keng[*]

December 4, 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**.

2 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

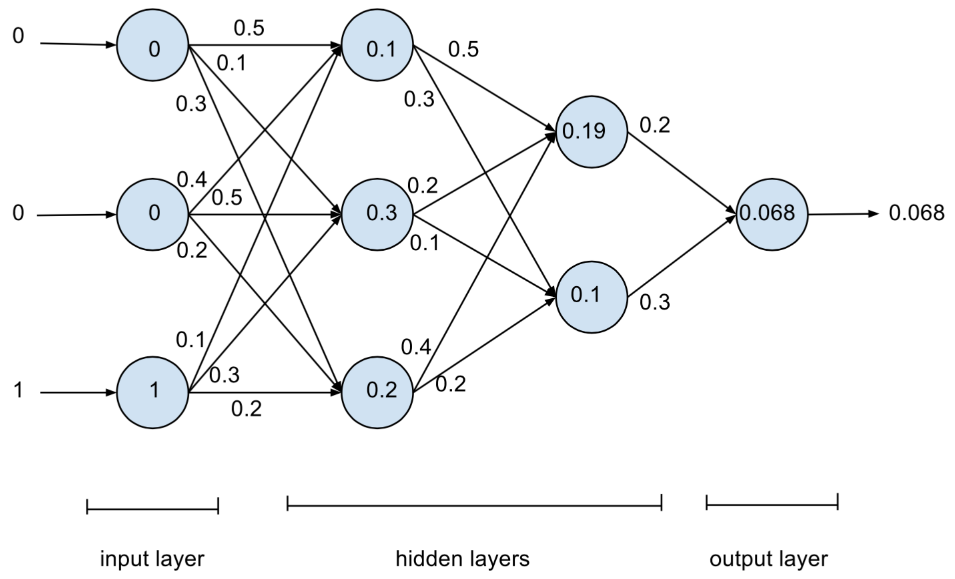

Figure 1: An ANN with 3 input units, 2 hidden layers each with 3 and 2 hidden units, and 1 output unit. The target function is the dot product between the incident values from the previous layer and their edge weights.

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

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

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.

2.1 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.

2.2 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 $d \in \mathit{Data}$ 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 $\Delta w_i$ 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
                $$w_i \leftarrow w_i + \Delta w_i$$
```

2.3 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 $\delta_k$ 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.*

3 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.

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

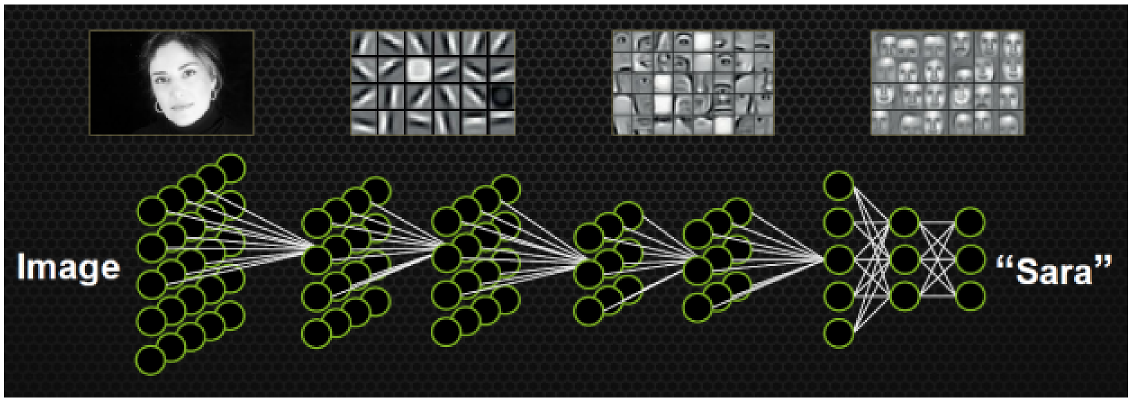

Figure 2: DNN that mimics the hierarchical structure of the brain. Going from left (input) to right, features get generalized from the specifics into more generic and abstract representations. At the top level (rightmost), `Image from NVIDIA.`

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.

3.1 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.

We now introduce the tensorial notation used for DNNs.

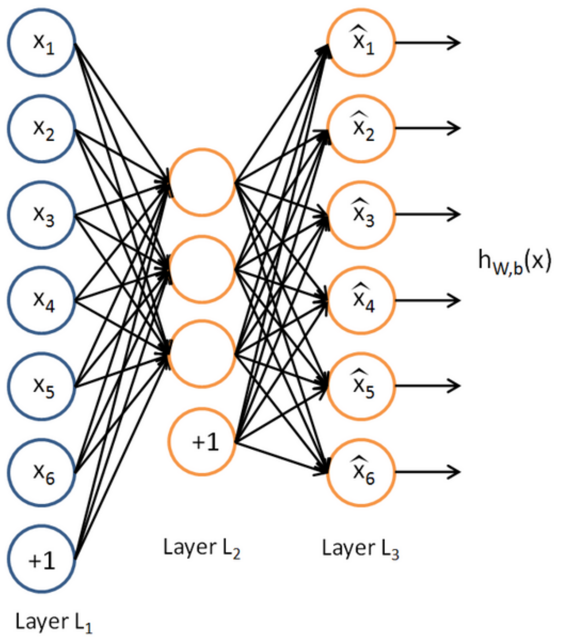

Figure 3: An autoencoder. The units with +1 are the bias units. The output units $\hat{x}_i$ will try to converge to x_i, as autoencoder mimics an identity function. `Image from UFLDL.`

3.2 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.

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

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.

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.

3.3 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.

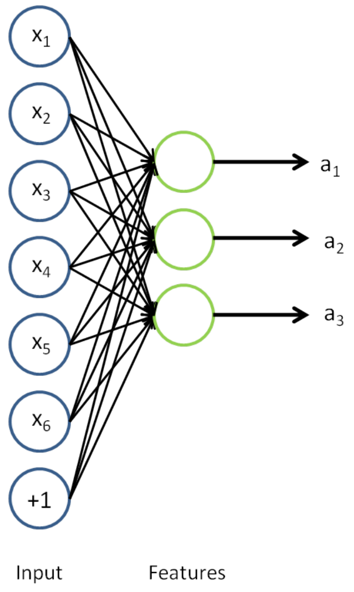

Figure 4: Self-taught learning: Pretraining Sparse Autoencoder - simply remove the third layer from the Autoencoder after training. `Image from UFLDL.`

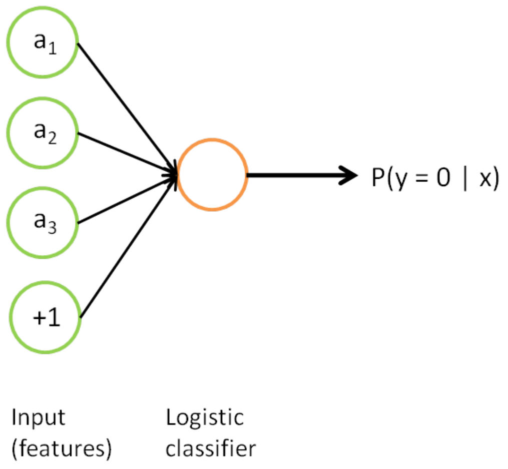

Figure 5: Supervised learning: Training Logistic Classifier (an ANN) using the activation units from the Sparse Autoencoder plus a bias unit. `Image from UFLDL.`

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.

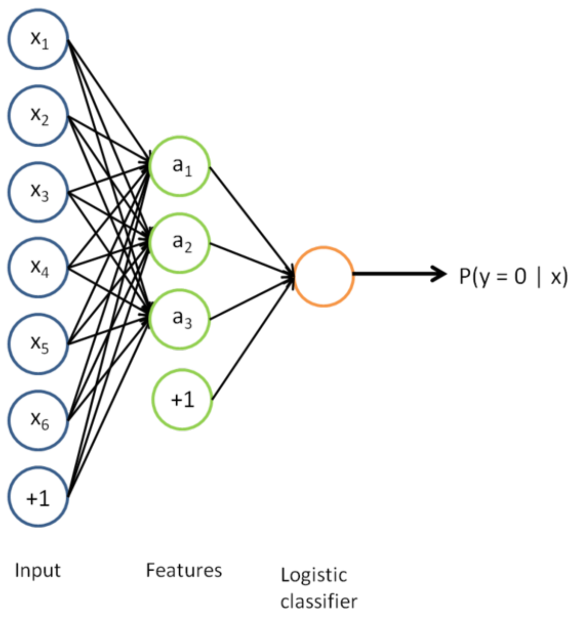

Figure 6: Semi-supervised learning as the basis for deep-learning: The network obtained by combining the Sparse Autoencoder from pretraining and the logistic classifier from supervised training. `Image from UFLDL.`

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.

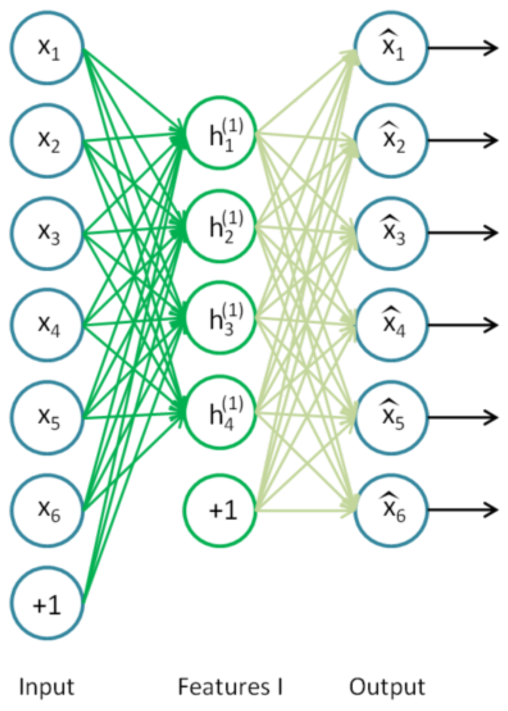

Figure 7: The first autoencoder, pretraining on the unlabeled data set with 6 input dimensions, and 4 hidden units. The pretraining will tune the edge weights to capture the first-order structure of the data. The output layer will then be discarded for stacking the next autoencoder. This process repeats for as many stacked autoencoders as desired. Once the pretraining is completely done, the last output layer is stripped for stacking a final classifier. `Image from UFLDL.`

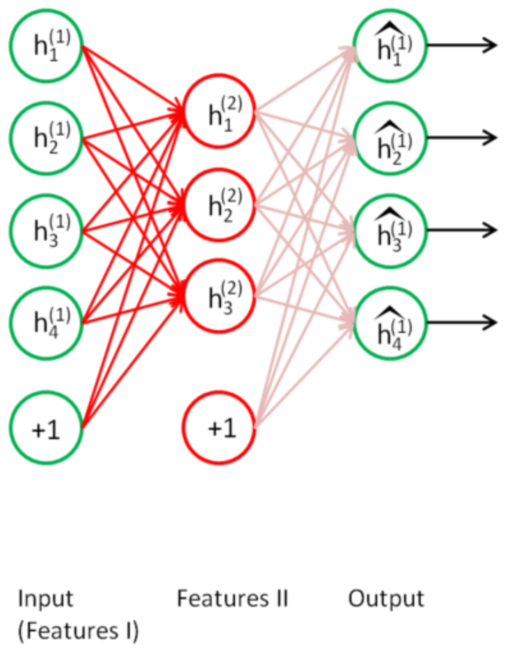

Figure 8: The second autoencoder. Then input layer is the hidden layer from the first autoencoder plus a bias unit. While pretraining this autoencoder, the edge weights for all the other layers are frozen. This autoencoder takes the feature output from the first and learns from it, thereby capturing the second-order structure of the data. In general, stacking an extra autoencoder involves using the feature output from the previous as input, adding a hidden and an output layer, and pretrain locally on the same unlabeled data set. The output layer is always removed afterward. `Image from UFLDL.`

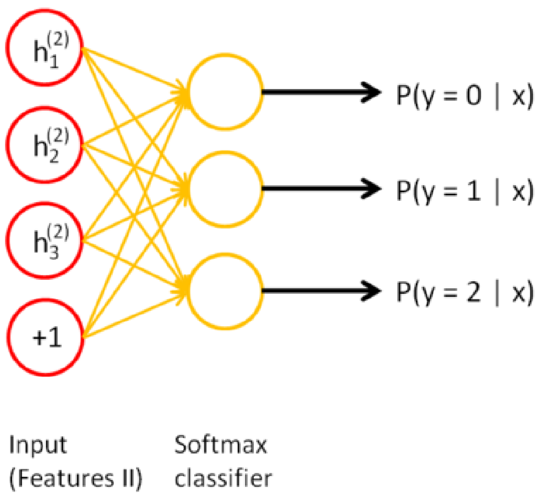

Figure 9: Pretraining is done for two hidden layers; the final (softmax regressor) 3-classifier is attached. `Image from UFLDL.`

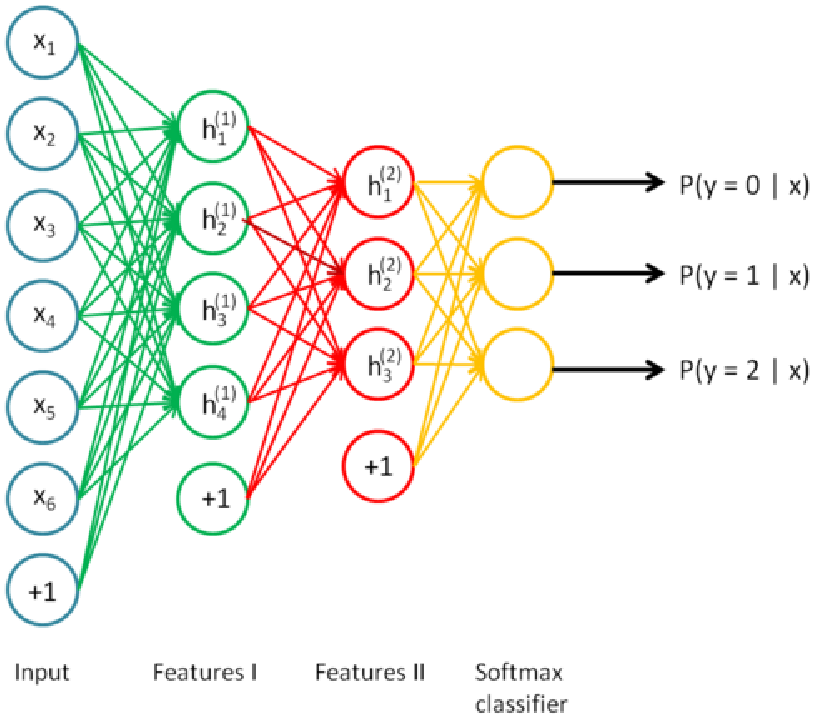

Figure 10: The resultant deep neural network of 2 hidden layers and a softmax classifier. Fine-tuning can be done on the whole network using backpropagation with gradient descent. The DNN takes in 6 input features $x_1, \cdots, x_6$ to predict the probabilities for 3 possible outcomes $\{0, 1, 2\}$. `Image from UFLDL.`

The are two popular yet simple choices of classifiers (ANNs that predict classes of outputs) - *logistic regression* and the *softmax regression* classifiers. The *logistic classifier* is used for outputs that are not mutually exclusive, such as the color and size of a cat. Whereas the *softmax classifier* is for mutually exclusive outputs, such as numerical digits.

3.4 Variants of Deep Networks

There are countless variants of deep networks, but their architectures are based off essentially the same ideas - make use of vast amount of data, and adding more hidden layers while still maintaining computational feasibility. They are primarily different in the training techniques, and the propagation of inputs across the network. We give a few of them with quick descriptions.

1. **Convolutional Neural Network (ConvNet):** Each layer tends to be arranged into a 2D matrix or pixels, and pooling is done - capturing the statistical properties of a submatrix (a section of the full pixels), such as the max value or the average, and passing it to a pixel on the next layer. This is particularly useful in computer vision, as pooling can capture many local structures such as colors and corners across the layers (such as the example shown in Figure 2).
2. **Recurrent Neural Network (RNN):** Unlike most neural nets that are feed-forward directed acyclic graphs (DAGs), RNN allows feedback cycles. This allows the output to contribute back to the next inputs, therefore has a strong temporal feature. It's is mostly used for handwriting and speech recognition, in which temporal order plays vital role in making sense of the data.
3. **Restricted Boltzman Machine (RBM):** This is a feedforward neural net that utilizes special energy function and training algorithm involving Gibbs sampling and Boltzmann distribution. The term "Restricted" implies that the hidden and visible layers must form a bipartite graph.

4 Applications

It is not an exaggeration to say that machine learning is everywhere today - we are living in the dawn of its golden age. Its rise can be attributed largely to:

1. *The invention of efficient algorithms and techniques.* Backpropagation, Self-taught learning, Greedy layer-wise training, and the likes due to Hinton, LeCun and their peers have lifted machine learning out from the theoretical dreamland into the practical world. As a result, many researchers and tech companies rode on the hype and helped propelled the movement further.

2. *The exponentiation of computational power and the decrease in cost.* Thanks to *Moore's law*, even our smartphones can be a node to train a deep network (as laid in in Google's *TensorFlow* discussed below).

3. *The abundance of data in the digital age.* This has made obtaining millions or billions of data points possible, and that is precisely why deep network became practical.

4. *Democratization of the technology.* The popularization of various machine learning APIs and open source libraries such as *IBM Watson, Alchemy, Metamind, Indico, ConvNet, cuDNN, Theano, Torch, TensorFlow* has made using and deploying machine learning close to trivial. They take care of the backend details, allowing for the developers to focus on applying the tools to various problems. Coincidentally, the invention of modern software engineering paradigm helps accelerates the pace of development - today, writing and deploying a full-stack web application while integrating a machine learning module takes just a day, whereas not too long ago it used to take up to weeks.

We next look at the major applications of machine learning throughout modern history.

When Google had its first try at creating artificial intelligence, they did not succeed. However, they did not fail exactly. The product in fact became Google Ad Sense, which powered (and still does) the majority of the company's revenue. This made Google's advertisement recommendation and placement system so advanced that it eclipsed all the other competitors. Given the encouraging outcome, Google would experiment more agressively with A.I. and machine learning in the time to come.

Decades ago linguists have tried to come up with the ultimate algorithm that captures Chomsky's idea of universal grammar - a simple, unifying formal grammar that will capture any human languages. The program had proven most challenging, when even the most respected linguists and computer scientists could not devise an algorithm that could parse and translate between different languages. Despite decades of attempts, we have not come close to realizing Chomsky's original proposition.

Then, Google started its own project, which would then become Google Translate. At that time, Google already had a sufficiently large amount of translated texts, for example between English and French. Instead of devising a collection of production rules and algorithms, the team stuck with an unusual approach - feeding the vast amount of translated texts they had into a neural net. Much to their surprise, it worked, and Google Translate was born. The result was much better than any carefully devised algorithms. No one knew how, but *it worked.* No wonder neural nets are often called a "black box", because there is nothing else other than the network and edge weights. If you feed enough data and teaches it about the output, it picks up on the pattern. Neural nets also help Google filter out 99.9% of the spams in Gmail.

2015 is a big year for machine learning - deep networks especially. Google showed off its deep learner that can quickly learn and play games on a super-human level. Google Deep Dream is another publicity stunt, in which an image recognizer spits out new psychedelic images when reversed - it "dreams" up images instead of recognizing. IBM Watson got its deep learning upgrade and learned to diagnose diseases and invent new cooking recipe. Facebook released M, their A.I. virtual assistant on the *Messenger* app. Weeks after, Google open sourced *TensorFlow*, their machine learning library. We will take a quick look at it.

4.1 TensorFlow

Released in November 2015 by Google, *TensorFlow*[6] is an API library for "Large-Scale Machine Learning on Heterogeneous Distributed Systems". Note that this by itself is not a new machine learning technique, but rather a new *architecture* that will streamline how we develop and research with machine learning.

TensorFlow is a culmination of Googlers' experience of deploying machine learning on a large scale. One can see it as a generalized, unified methodology for the commmunity. Generalization gives it flexibility for diverse applications; unification gives all the different team access to a well-maintained, consistent and up-to-date code base. It is already widely used at Google, and has helped many teams work faster and more efficiently.

At first glance, it is a tensor math library (Tensors are simply multidimensional array - scalars are rank-0 tensors, vectors are rank-1, matrices are rank-2, $\cdots$ and so on). It is equipped with many common and performant tensor operators for arithmetic, calculus, transformation, encoding, etc.

There is also many built-in popular machine learning components, such as softmax and logistic regressors, gradient descent and backpropagation algorithms, plain neural nets, and convolutional network with pooling.

4.1.1 Computation Graph

The truly novel feature of TensorFlow is its *programming model.* A TensorFlow computation is expressed as a directed graph - we call it the computation graph. A graph node is a computation step, or simply *Op* for *Operation.* This generalizes to the operation of updating state, inputing data and controling the logic. A directed graph edge represents the flow of data encoded in a *tensor*, from the output to the input node. There exists also special edges called *control dependencies* without data-flow - these merely specifies that the *source Op* must complete before the *target Op* can run.

An *Op* has a name such as *multiply, add*, and represents abstract computation. A *kernel* is a device-specific implementation of an operation. Device here refers to physical computation device such as the CPU or GPU. A *Variable* is a special kind of *Op* that allows for tensor-presistence during the runtime.

A *Session* instantiates an empty graph for contruction, and has the *Run* method to run the computation graph once construction is complete.

The data tensors that flow in the directed edges are intuitive to grasp. Tensors are generic and compact for discrete data-encoding; they are also natural to neural networks as we have seen their tensor formalizations before.

The graph-based operation and control flow offer a rich and intuitive programming model. The entire TensorFlow graph can be executed on a single device as usual, but things only get interesting when we consider a distributed execution.

A computation graph can be partitioned into disjoint subgraphs that are then placed on different devices. The overall distributed components, despite being separated, still remember their proper dependencies, thus the entire computation will remain coherent. This resource allocation is also handled automatically in the background. Furthermore, the model is fault-tolerant, as each subgraph can maintain state-persistence or perform independent computation even if some other subgraphs fail. The entire computation can then resume quickly by simply restarting the faulty components, while the healthy ones are always up and ready.

The subgraph-partitioning also allows for partial execution. For a large network, the user can develop and test a subgraph independently by providing special*feed* and *fetch* nodes that simulates data input and output for this subgraph. Such independence makes development easier as one can focus on a more manageable subgraph without having to warry about the larger graph. A subgraph can also be easily replaced by a new module, or be extended, as how one can easily extend a graph by adding edges and nodes.

4.1.2 Democratization

Distributed computing with graphs is not by itself anything novel. It is true that TensorFlow is not revolutionary in this sense, but this is not what Google try to do.

The primary contribution of TensorFlow is in *overcoming complexities* - it does not revolutionize machine learning, but it will certainly change *how we do* machine learning.

To illustrate, we show below how easy it is to use TensorFlow to train an ANN with softmax classifier to recognize images of digits.

```
import tensorflow as tf
# import the digit image data
```

```
import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

# create a new Session, then
# build the graph below by defining the nodes (Ops)
sess = tf.InteractiveSession()

# the input and output tensors of the ANN,
# shaped to match the image input dimension (pixels)
x = tf.placeholder("float", shape=[None, 784])
y_ = tf.placeholder("float", shape=[None, 10])

# The weights tensor and the bias tensor
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))

# initialize the Variables
init = tf.initialize_all_variables()

# run the session (graph) to initialize
sess.run(init)

# define the softmax model, also specifying
# the edges in the graph
# (e.g. Ops x, W, b and pointing to Op y)
y = tf.nn.softmax(tf.matmul(x,W)+b)
cross_entropy = -tf.reduce_sum(y_*tf.log(y))

# define the training step to use gradient descent
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)

# run the training on the data, over 1000 iterations
for i in range(1000):
        batch = mnist.train.next_batch(50)
        train_step.run(feed_dict={x: batch[0], y_: batch[1]})
```

In less than 20 lines of code, we can train an ANN to recognize images of digits at 98% accuracy - the simplicity achieved with TensorFlow is dramatic.

Just years ago, practical machine learning was the inaccessible toys of elite academics, mainly due to the implementations being highly technical and complex. Today, any interested person with some understanding of the subject can experiment and build her own machine learners. What TensorFlow has achieved/will achieve is the successful democratization of machine learning.

We look more carefully how TensorFlow democratizes machine learning:

1. *The choice of high-level interface language:* Tensorflow is implemented in `C++`, but the language that people interface with it is `Python`, which is a popular and easy-to-use scripting language. This makes everything much easier and concise, as we saw in the example above. More implementations will come soon in other popular languages as well.

2. *The programming model:* The usage of computation graph is robust and intuitive. The modularity that comes with it also allows users to focus on a much manageable subcomponent of a very large structure.

3. *Practicality:* The resilient distributed computing and efficient implementation makes the otherwise computation-heavy machine learning practical and fast, as most users can easily use it, and can divide the computation to different devices/resources such as the CPU, GPU of multiple connected machines. Individuals can use it even without owning an entire server farm.

4. *Scalability:* TensorFlow is built for the individuals as well as the engineering teams at Google. It can be a simple ANN that runs on a laptop, or a gigantic DNN that runs on some dedicated Google server farm. With the distributed computating, it can also run on thousands or millions of devices - like how Google is deploying DNN for Google Translate using the users' smartphones/computers in parts.

5. *Complexity-handling:* The various features described above - graph construction, dependencies, flow control, device/resource allocation - are all handled automatically in the background. Moreover, many useful *Ops* such as the gradient descend algorithm and the construction of a complete neural net are packaged into built-in functions. What would take hundreds of lines of code if self-implemented now takes only one line in TensorFlow.

6. *Open source:* The choice to release one of the most powerful proprietary tools of Google to the public isn't unreasonable. With contributions from the community, TensorFlow can improve and grow even faster - it's a win-win situation.

7. *Bridging the gap:* As part of the needs realized by the research groups at Google, "It is no longer acceptable to separate research from deployment," the company has been trying to shorten the transition between academic research and industrial application. TensorFlow answers the question by providing a unifying platform - researchers can propose and try out new ideas immediately on it, and if they work, send the same source code for the engineer to deploy.

At the time of writing, TensorFlow has only been released for less than a month. The reactions from the community have been welcoming, and people are rapidly picking up and experimenting with the tools. It will be some time before we can see the true impact of TensorFlow - whether or not it will dramatically push machine learning forward.

Hawkins CLA Andrew Ng, One learning algorithm

5 Future Work

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`
6. Dean, Jeffrey et. al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*, Google Research. 2015.