An Introduction to Deep Learning

Introduction

Modeling data via artificial neural networks (ANN) is not a new concept. Most of the underlying techniques have been known since the 1940s. It has to be pointed out though that a series of recent advances in how the networks are trained and utilized form the foundation of today's Deep Learning ecosystem. It is a fair statement that the recent advances brought forward by Deep Learning reflect a new era in Machine Learning (ML) that revolutionized many domains of signal and information processing. This holds true beyond the commonly discussed speech and object recognition science but also branches into computer vision, natural language processing, or information retrieval related fields.

Figure 1: Possible Stages of Developing a Deep ANN (Conceptional View)

Figure 1 discloses (conceptually) the phases necessary to develop a Deep ANN (DNN) that contains an input layer $x$, 3 hidden layers ($h_1$, $h_2$, and $h_3$), and an output layer $y$. The stages are labeled the pre-training, the fine-tuning, and the classification phase and such a network is labeled in the literature as a Deep Belief Network. At present, various forms of DNN’s are available such as Restricted Boltzmann Machines (RBM), Deep Belief Networks, or Deep Autoencoders. In a nutshell, a DNN represents a feed-forward neural network (a network does not form a cycle). A DNN maps the provided inputs (data or features) onto the required outputs (categories reflecting a behavior or a context). The network consists of nodes that are arranged into a series of fully connected layers. Sandwiched between the input and the output layers, the DNN is comprised of $n$ bridging layers (labeled the hidden layers). Simplified, each node utilizes an activation function to transform the data/state in the prior layer while the outcome gets exposed to the next layer. Some common node activation functions are sigmoid or tanh. A logistic sigmoid function $y = (1 / (1 + e^{-x}))$ has the property of returning values in the range $[0..1]$, which makes the function suitable to represent various probability scenarios. The output nodes are different as they typically use a softmax function where the final inference is determined via the node that holds the largest value (the conditional probability). A softmax function squeezes an $n$ dimensional vector $\tau$ that is holding some random real values into an $n$ dimensional vector $\sigma(\tau)$ that contains real values in the range $[0..1]$. A DNN is normally trained in 2 phases. First, an unsupervised process that is referred to as pre-training is invoked to bootstrap the hidden node and the edge parameters. This stage is a major milestone in Deep Learning.
Learning in general, as this phase can be effectively processed via a greedy, layer-wise approach that does not require any labeled data. The pre-training phase significantly simplifies learning when multiple hidden layers are present. Next, a supervised process is invoked that is labeled as the fine-tuning phase. In this phase, some form of backpropagation is applied to adjust the parameter values initialized in the pre-training stage. The parameters are adjusted to minimize the loss function that quantifies the delta between the network inferences and the actual labeled data.

It has to be noted that various variations of this training process have been proposed and that the DNN's themselves can be utilized in various ways to perform inference. To illustrate, DNN's can be used simply as classifiers, can get linked to interpret data of differing modalities, can be combined with other types of models (such as Hidden Markov or Gaussian Mixture Models) to form a hybrid, or can act as a frontend feature for a selection phase. Beyond the basic DNN approach, there is an entire group of models at the scientist's disposal such as Deep Belief Networks, Restricted Boltzmann Machines (RBM), Autoencoders, or Convolutional Neural Networks (to name a few). This paper further elaborates on RBM's and Autoencoders, respectively. In general, with supervised learning, a typical machine learning modus operandi would be like:

1. An ML algorithm is presented with some labeled examples such as 5,000,000 images that show a horse (label 1 = horse) and 5,000,000 images that represent something else (label 0 != horse). Ergo, this example focuses on a binary classifier.
2. The ML algorithm learns to identify the images that reflect a horse and when presented with a new image (unseen data), the goal is to generate the appropriate label (1 if the picture depicts a horse, 0 in all other scenarios).

Other examples could be a project where the data reflects patient symptoms and the label depicts a disease, or a case where the data shows images of handwritten characters and the labels mirror the actual characters they symbolize.

Perceptrons - The Building Blocks for many ANN's

Figure 2: Linear Classifier

A rather generic supervised training algorithm is known as the perceptron, which reflects a simple building block for artificial neural networks. Assuming there are n points in a plane that are labeled as either 0 or 1. The question at hand is presuming that a new data point is presented, how can that data point
be either labeled as 0 or 1? One approach may be to locate the nearest neighbor (distance based) and to return that data point’s label. A somewhat more sophisticated approach would be to introduce a line that best partitions the labeled data and use that line as the classification border. In this scenario, each input data point can be presented as a vector \( x = (x_1, x_2) \) and the function being used can be expressed as 'equal to 0 if below the line or equal to 1 if above the line' (see Figure 2). To represent this system mathematically, the separator can be defined as a vector of weights \( w \) and a vertical offset (or bias) \( b \). Hence, the (transfer) function combines the inputs and the weights with a weighted sum transfer function:

\[
f(x) = x^*w + b \quad \text{(Equation 1)}
\]

The result of the transfer function (Equation 1) can now be absorbed by an actual activation function (Equation 2) to generate a label. For the linear classifier, the activation function reflects a threshold cutoff scenario (the label is 1 if the result of the transfer function is greater than 0). Hence, the activation function can depicted as:

\[
h(x) := \begin{cases} 
1 & \text{if } f(x) > 0 \\
0 & \text{otherwise} 
\end{cases} \quad \text{(Equation 2)}
\]

The training phase of a perceptron consists of entering \( n \) training samples into the model and calculating the output for each of them. After each sample, the weights \( w \) are adjusted focusing on minimizing the output error. The output error is defined as the delta between the desired (the target) and the actual output values. Some of the more popular error functions being used in production are the mean squared error (MSE) or the mean percentage error (MPE). In many cases (especially for Deep Learning), a single perceptron based model is not sufficient to actually address a real-world problem, as a single linear perceptron can only learn a linearly separable function. To illustrate, a single linear perceptron cannot classify the output of a simple XOR function. In Figure 3, the output value 0 corresponds to the black dots while the output value 1 represents the white dots, respectively.

Figure 3: XOR Function

<table>
<thead>
<tr>
<th>( X_1 )</th>
<th>( X_2 )</th>
<th>( Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
Y = X_1 \oplus X_2
\]

In order to address even a simple problem such as an XOR function, it is normally necessary to utilize a multilayer perceptron (also known as feedforward neural networks). As a matter of fact, a composition of these simple perceptrons is normally used to spawn a more potent learning mechanism.
Feedforward Neural Networks & Deep Learning

To summarize, a neural network is basically just a composition of perceptrons that are connected in various ways and operate via different activation functions. A feedforward neural network essentially retains the following attributes:

- Holds an input, an output, and 1 or \( n \) hidden layers. Figure 4 depicts a network with a 3-unit input layer, a 3-unit hidden layer and an output layer that consists of a 2-unit setup (in the literature, the terms units and neurons are used interchangeably).
- Each unit represents a single perceptron.
- The input layer units depict the inputs into the hidden layer units while the hidden layer units represent the inputs into the output layer.
- Each connection between 2 neurons has a weight \( w \) (similar to the perceptron weights).
- Each unit of layer \( l \) is typically connected to every unit of the previous layer \( l - 1 \) (although one can disconnect them by setting their weight to 0).
- To process the input data, the input vector is mapped onto the input layer, placing the values of the vector as outputs for each of the input units. Based on Figure 4, the network can process a 3-dimensional input vector (because the network consists of 3 input units). To illustrate, for an input vector \([9, 4, 5]\), the output of the top input unit would be set to 9 while the middle output would be reflected via the value 4 and the bottom output would take on the value 5. In a next step, these values are propagated forward to the hidden units while utilizing the weighted sum transfer function for each hidden unit. The hidden units calculate their outputs via the provided activation function.
- The output layer calculates the output values in the same manner as the hidden layer. The result of the output layer reflects the output of the network (a 2-term vector \([x, y]\) based on Figure 4).

If each perceptron operates on a linear activation function, the final output of the network still reflects some linear function of the inputs that is just adjusted via a range (set) of different weights that are collected throughout the network. Hence, a linear composition of a set of linear functions still just represents a linear function. Therefore the statement can be made that if a feedforward neural network model is restricted to linear activation functions, the model is no more powerful than a perceptron, no matter how many layers are used. Ergo, most neural networks are based on nonlinear activation functions such as \( \tanh \), \( \text{sigmoid} \), \( \text{rectifier} \), or \( \text{softplus} \) (a smooth approximation to the rectifier - see Figure 5).

A very common (deep) learning algorithm for multilayer perceptrons is known as backpropagation. The basic backpropagation process can be described as:

- A training sample is presented to the model and propagated in a forward manner through the network.
- The output error is calculated (such as the MSE): \(MSE = \frac{1}{2} (t - y)^2\) (Equation 3)
- In Equation 3, \(t\) represents the target value while \(y\) depicts the actual network output. Note: Equation 3 is valid for a single output neuron model. The factor \(1/2\) is included to cancel out the exponent when differentiating. Later, the expression is multiplied with an arbitrary learning rate so it does not matter if a constant coefficient is introduced (such as in Equation 3).
- The actual network error is minimized by applying a stochastic gradient decent method.

While gradient descent reflects a universal method, in conjunction with ANN's, gradient decent depicts a graph of the training error as a function of the input parameters. The optimal value for each weight is the value at which point the error reflects the global minimum (see Figure 6). During the training phase, the weights are updated in small increments/steps (either after each training sample or after a batch of \(n\) samples). The goal is to reach the global minimum (represented by \(b\) in Figure 6), but it has to be pointed out that this is no easy task as it is not unusual to end up in a local minima (such as the one depicted by \(d\) in Figure 6). To further illustrate, in Figure 6, if the weight has a current value of \(b + x\) (\(b + x < c\)), the weight value needs to be adjusted so that the value moves towards \(b\). Figure 6 basically depicts the simplest scenario where the error only depends on a single parameter. However, the actual network error depends on every network weight and hence the error function itself is much more complex. Luckily, backpropagation provides a method for updating each weight between 2 neurons with respect to the output error. The derivation in itself is rather complicated but the weight update for a given node follows the form depicted in Equation 4 (the character \(\partial\) references the partial derivative while \(\alpha\) depicts the learning rate).

\[
\Delta \omega_i = -\alpha * (\partial E / \partial \omega_i) \quad \text{(Equation 4)}
\]

In Equation 4, \(E\) represents the output error while \(\omega_i\) depicts the weight of input \(i\) to the neuron. Fundamentally, the goal is to move in the direction of the gradient with respect to the weight \(i\). The key
term here is the derivative of the error, which is not always easy to calculate. In other words, the question is how to identify the derivative for a random weight of a random hidden node in the middle of a large network? The answer to the question is again via backpropagation. To illustrate, the errors are first calculated at the output units where the formula is rather simple (basically the delta between the target and predicted values has to be calculated) and then are propagated back through the network. This allows to efficiently update the weights during the training phase and to (hopefully) reach the global minimum.

Figure 6: Local & Global Maximum/Minimum

It has to be noted that the hidden layer is of particular interest here. Based on the universal approximation theorem (which simplified states that basic neural networks can represent a wide variety of appealing functions when given appropriate parameters), a single hidden layer network with a finite number of neurons can be trained to approximate an arbitrarily random function. In other words, a network model that consists of a single hidden layer is sufficient to learn any function. That said, one sometimes learns better in the real world if multiple hidden layers (deeper nets) are available. The hidden layer is where the network stores its internal (abstract) representation of the training data, similar to the way that the human brain (greatly simplified) contains an internal representation of the real world.

A neural network can obviously consist of more than 1 hidden layer. In such a scenario, the additional layers are constructing new abstractions on top of the previous layers. Based on the problem at hand, it is sometimes beneficial to learn in larger network setups. However, increasing the number of hidden layers may trigger some nasty scenarios:

1. The Vanishing Gradients issue is due to the fact that as more hidden layers are introduced, backpropagation becomes less useful in passing information to the lower layers. As a matter of fact, as the information is passed back, the gradients start to vanish and become small relative to the weights of the network.
2. The Overfitting issue basically reflects the innermost problem with machine learning. In a nutshell, overfitting describes the phenomenon of fitting the training data too closely. In other words, the model learns the training data and not the mapping and hence does not really generalize.

To address these issues, several deep learning algorithms are available and the Restricted Boltzmann Machines and the Autoencoders are further elaborated on in the next few paragraphs.
Autoencoder

An autoencoder basically reflects a feedforward neural network that focuses on learning a compressed, distributed representation (encoding) of a dataset. In Figure 7, the circles labeled "+1" are called the bias units and correspond to the intercept term. The intercept (the constant) is the expected mean value of $Y$ when all $X = 0$.

Figure 7: Autoencoder

Conceptually, the network is trained to recreate the input or in other words, the input and the target data are the same. So the objective is to output the same as the input but in a compressed fashion. The best way to further explain an autoencoder is via some examples. Assuming a scenario where the training data consists of 28x28 grayscale images. The value of each pixel (28x28 = 784) is presented to an input layer unit and hence the input layer consists of 784 neurons. As the input layer consists of 784 neurons, as the example discusses an autoencoder, the output layer also consists of 784 units. The target value for the units are grayscale values for each pixel. The objective here is not to learn the mapping per se but to learn the internal structure and features of the data itself. With autoencoder networks, the hidden layer is also referred to as the feature detector. In most scenarios, the number of hidden units (features) is smaller than the number of input/output neurons, which forces the network to learn only the most important features and so also achieves a dimensionality reduction. Again, the objective is that a few hidden units learn the data at a conceptual level and produce a compact representation of the data that captures the core features of the input.

To further elaborate on the working of autoencoders, the following example discusses a dataset that contains flu symptoms information (this example is motivated by a case study presented by Vasilev). The data vector consists of 6 binary input features $[a \ b \ c \ d \ e \ f]$ where the first 3 features $[a \ b \ c]$ refer to the symptoms of the flu whereas the last 3 features $[d \ e \ f]$ describe counter symptoms. To illustrate, an input...
vector of \([1\ 0\ 0\ 0\ 0\ 0]\) indicates that the patient has a fever, \([0\ 1\ 0\ 0\ 0\ 0]\) refers to the fact the patient has a headache, whereas \([0\ 0\ 1\ 0\ 0\ 0]\) points out that the patient is congested. When some of the counter symptoms are present, the patient is less likely to have the flu. To illustrate, an input vector of \([0\ 0\ 0\ 1\ 0\ 0]\) depicts the fact that the patient had a flu shot, \([0\ 0\ 0\ 0\ 1\ 0]\) indicates that the patient is taking in additional vitamin D, whereas \([0\ 0\ 0\ 0\ 0\ 1]\) refers to a patient who is eating fermented food. It is possible to have a feature vector that contains elements of both sets such as \([0\ 1\ 0\ 1\ 0\ 0]\) that refers to a patient that has a headache and has received a flu shot. In this example, a patient is considered to have the flu if he or she shows at least 2 of the of the first 3 symptoms and is labeled as being healthy if he or she matches at least 2 of the 3 last features (the tiebreaker goes to healthy in this example). Therefore some of the possible input vectors and labels could be:

- has the flu -> 111000, 101000, 110000, 011000, 011100
- is healthy -> 000111, 011110, 000101, 000011, 000110

To address this problem via the usage of an autoencoder (and using backpropagation), a network model with 6 input and 6 output units is established. The network has 2 hidden units, referring to either the label 'has the flu' or 'is healthy'. What happens is that after presenting the network with a few thousand input vectors, depending on the feature combination presented to the model, either the 'has the flu' or the 'is healthy' hidden unit exhibits a higher activation rate than the other. In other words, at that point, the 2 hidden units have learned a compact representation of the flu symptom dataset.

**Restricted Boltzmann Machines**

Next to the autoencoders introduced in the last few paragraphs, the discussion now focuses on Restricted Boltzmann Machines (RBM). In a nutshell, an RBM represents a generative stochastic neural network capable of learning a probability distribution over its set of inputs. RBMs are composed of a hidden, a visible, and a bias layer (see Figure 8). Unlike feedforward networks, the connections between the visible and the hidden layers are undirected. In other words, the values can be propagated in both directions and the networks are fully connected (each unit in a given layer is connected to each unit in the next layer). Networks where any unit in any layer is connected to any other layer are referred to as Boltzmann machines and not as restricted Boltzmann machines.

A typical RBM implementation consists of binary hidden and visible units where the unit activation is either 0 or 1 under a Bernoulli distribution (a probability distribution of a random variable that takes on a value of 1 with a success probability of \(p\) or a value 0 with a failure probability \(q = 1 - p\)). While the AI community has worked with RBM's for quite some time, the recent emergence of the *contrastive divergence* unsupervised training algorithm used for some Deep Learning projects has significantly boosted the interest. A single-step contrastive divergence algorithm consists of 2 phases:

- A positive phase where an input sample \(v\) is presented to the input layer. The input sample \(v\) is propagated to the hidden layer similar to what is being done with feedforward networks. The result of the hidden layer activation process is labeled as \(h\).
- A negative phase where \(h\) is propagated back to the visible layer generating a result \(v'\). To reiterate, with RBM's, the connections between the visible and the hidden layers are unidirectional and hence allow data movements in both directions. Next, the new result \(v'\) is propagated back to the hidden layer with activations that result in generating \(h'\). The weight update is governed by:
  \[ o(t + 1) = o(t) + \alpha(v'h' + v'h) \] (Equation 5)
In Equation 5, \( a \) represents the learning rate whereas \( v, v', h, h' \) and \( w \) reflect the vectors. The impetus here is that the positive phase (\( h \) given \( v \)) depicts the network’s internal representation of the real world data. Meanwhile, the negative phase represents an attempt to recreate the data based on the internal representation (\( v' \) given \( h' \)). The objective is for the generated data to be as close as possible to the real world, which is reflected in the weight update term (Equation 5). In other words, the network has some sensitivity of how the input data can be represented and the objective is to reproduce the data based on this perception. If the reproduction is not close enough to the real world, the network makes adjustments and tries again.

Figure 8: Restricted Boltzmann Machine

To further elaborate on RBM's and contrastive divergence, the discussion revisits the flu symptoms dataset discussed earlier. To address the problem, an RBM network with 6 visible and 2 hidden units is established. The network is trained via contrastive divergence and the 6 symptoms represented by vector \( v \) are presented to the visible layer. During the test phase, the symptoms are again presented to the visible layer and the data is propagated to the hidden layer. The hidden units represent the 'has the flu' and the 'is healthy' states. The statement can be made that the RBM and the autoencoder architectures are similar. With the RBM network model, after a few thousand iterations, similar results as with the autoencoder are observed where 1 of the 2 hidden units shows a higher activation rate depending on what symptoms are presented.

Deep Learning (Belief) Networks

So far, the discussion focused on the fact that the hidden layers of either autoencoder or RBM networks act as effective feature detectors. The issue though is that in practice, it is rather unusual that these features can be directly used. In most scenarios, it is required to identify some way to use these detected features indirectly. Hinton et al. (University of Toronto) discovered that these structures can be stacked to form deep networks. These networks can be trained in a greedy manner (1 layer at a time) to aid in overcoming the vanishing gradient and overfitting problems that are associated with a classic backpropagation based approach. The resulting structures are rather potent and may produce remarkable results on unlabeled data (such as with the Google cat image project).

Stacked Autoencoders

To illustrate, the hidden autoencoder layer \( l \) serves as an input layer into another autoencoder \( l + 1 \). The input layer of the first autoencoder reflects the input layer into the entire network. With stacked autoencoders, the greedy, layer-wise training circle can be described as:

1. Train the first autoencoder (\( l=1 \), which is depicted by the green connections in Figure 9) with an additional output layer by utilizing the backpropagation method and all the training data.
2. Train the second autoencoder \( l=2 \) (red connections). As the input layer for \( l=2 \) depicts the hidden layer \( l=1 \), there is no interest in the output layer of \( l=1 \) and hence that input layer is removed.
from the network. The actual training commences by providing an input sample to the input layer $l=1$, which is propagated forward to the output layer of $l=2$. Next, the weights (input-hidden and hidden-output) of $l=2$ are updated via backpropagation. It has to be noticed that $l=2$ (similar to $l=1$) utilizes all the training data.

3. Repeat the previous practice for all the layers. In other words remove the output layer of the previous autoencoder, replace it with yet another autoencoder and train the network via backpropagation.

![Stacked Autoencoder](image)

The above steps (1 through 3) are labeled the pre-training phase and they leave the weights properly initialized. However, there is no mapping between the input data and the output labels. To illustrate, if the network is trained to recognize images of handwritten letters it is still not possible to map the units from the last feature detector (the hidden layer of the last autoencoder) to the actual letter type of the image. To resolve this, the most universal approach is to add 1 or $n$ fully connected layer(s) to the last layer (the yellow connections). Now, the entire network can be viewed as a multilayer perceptron that can be trained via backpropagation (this last step is labeled the fine-tuning phase). In a nutshell, stacked autoencoders provide an effective pre-training method for initializing the weights of a network that after the pre-training phase provides a complex, multi-layer perceptron model that is ready to be trained (or fine-tuned).

**Stacked Restricted Boltzmann Machines**

Similar to the discussion on autoencoders, it is feasible to utilize restricted Boltzmann machines (RBM) to fashion a class of networks that are labeled as deep belief networks (DBNs). With RBM’s, the hidden layer $l$ acts as the visible layer for the RBM $l+1$ (see Figure 10). The input layer of the first RBM reflects the input layer for the entire network, and the RBM based greedy, layer-wise pre-training cycle can be expressed as:
1. Train the first RBM \((l=1)\) via contrastive divergence while using all the training data.

2. Train the second RBM \((l=2)\). As the visible layer for RBM \(l=2\) represents the hidden layer of RBM \(l=1\), the training starts by presenting an input sample to the visible layer of \(l=1\), which is propagated in a forward fashion to the hidden layer of RBM \(l=1\). This data basically serves as the initiating entity for the contrastive divergence training phase for RBM \(l=2\).

The steps outlined above are repeated for all the layers in the model. Similar to the discussion on stacked autoencoders, after the pre-training phase, the network can be extended by connecting 1 or \(n\) fully connected layers to the last (RBM) hidden layer. By doing so, the network forms a multi-layer perceptron that now can then be fine tuned via backpropagation. This process is analogous to the approach taken with stacked autoencoders except that in the pre-training phase, autoencoders utilize backpropagation while RBM's rely on the contrastive divergence algorithm.

Figure 10: Stacked RBM
Case Study I - Movie Suggestions - RBM

The assumption made for this case study is that a market analysis firm is tagged by an online movie company to conduct a survey to rate a set of movies on a scale from 0 to 100. By applying a classical factor analysis evaluation (a factor analysis is used to describe variability among observed, correlated variables in terms of a potentially lower number of unobserved variables called factors) the firm could elaborate on each movie and user in terms of a set of latent factors. Latent factors reflect variables that are not directly observed but are rather inferred (through a mathematical model) from other variables that are observed (directly measured). To illustrate, movies such as *The Man Who Shot Liberty Valance* or *Once Upon a Time in the West* disclose a strong associations with a latent Western factor while users who like *The Incredibles* or *Ratatouille* reveile a strong associations with a latent Pixar Animation factor. Note: This case study is derived from E. Chen's discussion on Restricted Boltzmann Machines.

It can be pointed out that Restricted Boltzmann Machines can effectively be used to depict a binary description of a factor analysis. There are obviously varies other ways to utilize RBM’s, but for this case study, the RBM is used as the fundamental framework. Hence, instead of the users rating a set of movies on a continuous scale, the assumption made is that the users stipulate either if they like or dislike a move. Based on the collected data, an RBM model can be used to discover latent factors that explain the activation of these movie choices. To reiterate, a Restricted Boltzmann Machine reflects a stochastic neural network. The term neural network refers to the neuron-like units that are linked while the network uses transfer and activation functions. The term stochastic implies that these activations contain a probabilistic element. A RBM consists of:

- A layer that contains the visible units (the users’ known movie preferences)
- A layer that holds the hidden units (the latent factors that the RBM tries to learn)
- A bias unit (this unit is always on and is used to adjust the movie's popularity gage).

Each visible unit is connected to all the hidden units and all connections are undirected. The bias unit is connected to (1) all visible units and (2) all hidden units, respectively. To ease the learning process, neither the set of visible nor the set of hidden units are connected within the set. If they would be, the network would be known as a Boltzmann machine and not a restricted Boltzmann machine. To further illustrate the example, the case study is based on a set of 6 movies. The 6 movies are *Tombstone, True Grit, Open Range, The Rock, Bull Durham,* and *Casablanca.* In the survey, each user in the sample population is asked to select the movies that he/she would like to watch. The objective is to learn 2 latent units that underlie the movie preferences. Based on the 6 movies, 2 natural groups in the set of 6 movies are *Western Movies* (*Tombstone, True Grit,* and *Open Range*) and *Oscar Winners* (*Casablanca, True Grit,* *The Rock*). The resulting RBM model is depicted in Figure 11.

Neural networks basically operate by updating the states of some neurons given the states of other neurons. Hence it is paramount to understand how the states of individual units actually get adjusted. Assuming that the connection weights in an RBM are known (explanation below), to update the state of unit $i$, the following process can be outlined:

- Compute the activation energy $a_i = \sum j w_{ij} x_j$ of unit $i$, where the sum is compiled across all the units $j$ that unit $i$ is connected to. Further, $w_{ij}$ represents the connection weight between $i$ and $j$ while $x_j$ reflects either the 0 or 1 state of unit $j$. In other words, all the neighbors of unit $i$ submit a message to $i$ and the system compiles the sum of all these messages.

Let \( p_i = \sigma(a_i) \) where \( \sigma(x) = \frac{1}{1+e^{-x}} \) reflects the logistic function. It has to be pointed out that \( p_i \) is close to 1 for large positive activation energies and close to 0 for negative activation energies.

The system activates unit \( i \) with probability \( p_i \) and deactivates the unit with probability \( 1 - p_i \). Simplified, units that are positively connected to each other seek to share the same state (on or off) while units that are negatively connected to each other prefer to be in different states.

Figure 11: RBM - Movies

To further exemplify the discussed process, the assumption made in this case study is that the 2 hidden units reflect Western and Oscar Winner movies, respectively. If Amy answered the survey and provided her six (binary) preferences, the RBM model can now be invoked to determine which of the 2 hidden units her preferences activate. In other words, the RBM explains her preferences in terms of the latent factors. The 6 movies (visible units) all submit messages to the 2 hidden units notifying them to imitate an internal update. It has to be pointed out that even if Amy answered that she would like to see Tombstone and True Grit, her selection does not guarantee that the Western hidden unit really gets activated, but only states that the Western unit has a high probability to get activated.

If a user declared in the survey that she would like to watch Tombstone and True Grit, the assumption can be made that she likes Western movies in general, but there is obviously a probability that she picked the 2 movies for some other reason (maybe she just likes the lead actor in these movies). Anyhow, based on this movie selection scenario, the statement can be made that the RBM depicted in Figure 11 generates models of people's opinions in a cluttered, convoluted, real world fashion. On the other hand, if it is known that a person likes Western movies (aka the Western hidden unit is active), the RBM can be used to determine which of the movie units (visible units) that hidden unit has active and so generate a set of movie recommendations for that user. In this scenario, the process is reversed as the hidden units submit messages to the visible (movie) units instructing them to update their respective states.

The question now is how does the system learn the connection weights in the network? To illustrate this step, the assumption made is that many training examples are available and that each training sample reflects a binary vector with 6 entities that correspond to a user's movie preference feedback (such as \([1,0,1,0,0,0]\)). For each epoch, the following update procedure is executed:
1. Obtain the 1st training example (such as \([1,0,1,0,0,0]\)). Set the states of the 6 visible units to these values (such as Tombstone=1, True Grit=0, Open Range=1, The Rock=0, Bull Durham=0, Casablanca=0).

2. After that, update the states of the hidden units by using the logistic activation rule already discussed in this case study. In other words, for the \(j\)th hidden unit, compute its activation energy \(a_j = \sum_i w_{ij} x_i\), and set \(x_i\) to 1 with a probability \(\sigma(a_j)\) and to 0 with a probability \(1 - \sigma(a_j)\). Next, for each edge \(e_{ij}\), compute the Positive\((e_{ij})\) = \(x_i * x_j\) value (for each pair of units, basically measure whether or not both are active).

3. Now reconstruct the visible units in a similar fashion. So for each visible unit, compute its activation energy \(\alpha_i\) and update its state (it has to be pointed out that this reconstruction process may result into not matching the original preferences). After that, update the hidden units again and compute the Negative\((e_{ij})\) = \(x_i * x_j\) value for each edge.

4. Next, update the weight of each node/edge \(e_{ij}\) by calculating \(w_{ij} = w_{ij} + \alpha * (Positive(e_{ij}) - Negative(e_{ij}))\). In this formula, \(\alpha\) depicts the learning rate.

Repeat the above update procedure for all the training data. In general, while working with machine learning algorithms, such a process continuous until a system converges, which implies that either the error among the training examples and their reconstruction values is below a certain threshold or that the system reached a (predefined) maximum number of epochs. The update procedure discussed above has a lot of merit. In step 2, Positive\((e_{ij})\) quantifies the association between the \(i\)th and \(j\)th unit that one intends the network to learn based on the training examples. Further, in step 3 (reconstruction) where the RBM generates the states of the visible units based on the hypotheses established for only the hidden units. Negative\((e_{ij})\) determines the association that the network itself generates (or daydreams about) when no units are fixed to the training data. Hence, by adding the result of Positive\((e_{ij})\) − Negative\((e_{ij})\) to each edge weight, the RBM runtime environment aids the network to better match reality (as depicted via the training data). While this update rule is referred to in the literature as contrastive divergence, a more appropriate description would be approximate gradient descent. For this case study, an actual RBM model was implemented in C++. The data set used for this project was generated by incorporating a pseudo random number generator into the project framework. The generated training data basically resembles the set shown in Table 1.

Table 1: Survey Results

<table>
<thead>
<tr>
<th>User</th>
<th>Tombstone</th>
<th>True Grit</th>
<th>Open Range</th>
<th>The Rock</th>
<th>Bull Durham</th>
<th>Casablanca</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Western</td>
</tr>
<tr>
<td>Colt</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Oscar Winners &amp; Bull Durham</td>
</tr>
<tr>
<td>Sue</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Western</td>
</tr>
<tr>
<td>Jeff</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Western &amp; The Rock</td>
</tr>
<tr>
<td>Pat</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Western</td>
</tr>
<tr>
<td>Rich</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Oscar Winners</td>
</tr>
<tr>
<td>Katy</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Western, not Open Range</td>
</tr>
<tr>
<td>Todd</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Oscars Winners</td>
</tr>
<tr>
<td>Abby</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Western</td>
</tr>
<tr>
<td>Tom</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Oscar Winners, not True Grit</td>
</tr>
</tbody>
</table>

After running the network through 10,000 test samples, the following weights emerged:

---

Table 2: Network Model Output

<table>
<thead>
<tr>
<th></th>
<th>Bias Unit</th>
<th>Hidden 1</th>
<th>Hidden 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias Unit</td>
<td>-0.42315601</td>
<td>-0.37004433</td>
<td>-0.8203297</td>
</tr>
<tr>
<td>Tombstone</td>
<td>1.48014766</td>
<td>-4.61098323</td>
<td>2.67833415</td>
</tr>
<tr>
<td>True Grit</td>
<td>3.02778376</td>
<td>0.54716751</td>
<td>2.0728677</td>
</tr>
<tr>
<td>Open Range</td>
<td>0.26248893</td>
<td>-3.97033594</td>
<td>1.14450026</td>
</tr>
<tr>
<td>The Rock</td>
<td>-0.1076228</td>
<td>3.83305032</td>
<td>-1.33174103</td>
</tr>
<tr>
<td>Bull Durham</td>
<td>-3.33956581</td>
<td>-0.55349871</td>
<td>-2.33040109</td>
</tr>
<tr>
<td>Casablanca</td>
<td>-1.33810015</td>
<td>4.54049058</td>
<td>-3.17934584</td>
</tr>
</tbody>
</table>

Based on the results, the statement can be made that the 1st hidden unit reflects the Oscar Winners (True Grit, The Rock, and Casablanca are active) while the 2nd hidden unit represents the Western movies (Tombstone, True Grit, and Open Range are active). It has to be pointed out that the 2 hidden units are not static, or in other words, while running the network model again, hidden unit 1 may correspond to the Western movies and hidden unit 2 to the Oscar winners, respectively. To further assess the sample RBM network, 4 more users were introduced and after the RBM model was trained (see Table 2), their preferences were utilized to benchmark (test) what hidden unit the RBM model would activate (the results are depicted in Table 3). As the network activates (in most cases) the right hidden unit, such an RBM model could be used to suggest movies to people who either favor Oscar winners or Western movies.

Table 3: Additional Users - Test Phase

<table>
<thead>
<tr>
<th>User</th>
<th>Preference</th>
<th>Model Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debby</td>
<td>All 3 Western</td>
<td>0,1</td>
</tr>
<tr>
<td>Bret</td>
<td>2 Western</td>
<td>0,1</td>
</tr>
<tr>
<td>Teddy</td>
<td>All 3 Oscar Winners</td>
<td>1,0</td>
</tr>
<tr>
<td>Amber</td>
<td>2 Oscar Winners</td>
<td>1,0</td>
</tr>
</tbody>
</table>

Note: The model output refers to which Hidden Unit is activated (referenced by 1). So a setting of 0,1 refers to having Hidden Unit 1 deactivated while Hidden Unit 2 is active (see Table 2). Hence, Debby’s preferences activated Hidden Unit 2 (Western) while Teddy’s choices activated Hidden Unit 0 (Oscar Winners).

In this introductory RBM discussion, \( \text{Negative}(e_{ij}) \) was quantified via the product of the \( i \)th and \( j \)th units after reconstructing the visible units once and then updating the hidden units again. For better results, it would be feasible to calculate the product after \( n \) number of reconstruction steps. In other words, cycle through updating the visible units, then the hidden units, then the visible units again, then the hidden units again and so on until the system reaches \( n \). While this approach is slower in practice, it better describes the daydreaming behavior of the network. Further, while the current model is based on \( \text{Positive}(e_{ij}) = x_i \cdot x_j \) where \( x_i \) and \( x_j \) represent binary states (0 or 1), the system could be developed so that \( x_i \) and \( x_j \) reflect activation probabilities (the same statement holds true for \( \text{Negative}(e_{ij}) \) term). Another optimization technique that could be considered is penalizing larger edge weights to obtain a sparser or more regularized model. When updating the edge weights, the system could use a momentum factor. To illustrate, for each edge, the model could insert a weighted sum of the current step (a \( \alpha \) * (\( \text{Positive}(e_{ij}) \) – \( \text{Negative}(e_{ij}) \) term) in conjunction with the step that was previously taken. From a performance perspective, it is feasible to use a set of \( n \) input data vectors (a batch compared to using just 1 in each epoch). With such an approach, the model would only update the network weights on a per batch basis, which in practice usually expedites the learning process due to leveraging swift matrix-multiplication algorithms.

Case Study II - Image Analytics - RBM

Image analytics describes the automatic algorithmic extraction and logical analysis process for information embedded into image data. The process itself is based on digital image processing techniques. The familiar bar codes and QR codes are simple examples that are used by most people today. The real image challenges though cover a vast spectrum ranging from facial recognition to movement analysis. Today, images and image sequences (videos) reflect ~80% percent of the corporate and public unstructured Big Data pool. As the growth of unstructured data keeps going, analytical systems are needed that are capable of assimilating and interpreting images and videos, as well as other data such as text and numbers. In a nutshell, an image is basically a set of signals that are sensed by the human eye and processed by the visual cortex in the brain, creating a vivid experience of a scene that is instantly associated with concepts and objects previously perceived and recorded in one’s memory. To a computer system, images are either a raster or a vector image. Simplified, raster images depict a sequence of pixels with discrete numerical values for color while vector images represent a set of color-annotated polygons. To perform analytics on images or videos, the geometric encoding has to be transformed into constructs that depict physical features, objects, and movement represented by the image or video. These constructs can then be logically analyzed on computer systems.

Many large firms (such as Google, Baidu, or Facebook) utilize actual Deep Learning models for various Machine Learning tasks. The brunt of the work focuses right now on speech and image recognition related projects, as well as to a lesser extent on natural language processing (NLP - human computer interaction). As already discussed, the Deep Learning paradigm focuses on stacking various types of transformers (the layers). Each layer operates on a set of input values (labeled as activations where each value corresponds to the firing rate of the neurons) and transforms the input into a different set of values via some internal parameters (the trainable synapses). In its most basic setup, the first set of activations represents the input while the last set reflects the probabilities of the input set belonging to n individual classes. During the training phase, the system provides the network with a set of input examples and the network adjusts its parameters (learns the pattern) to transform the inputs into proper class probabilities. For this case study, the assumption being made is that there is an image of let's say a 28x28 array of pixel values that represent the number 5. Next, a system can be setup that contains a 2D representation of size 28x28 containing the pixel values. The system can now present the input set to an (RBM) network that generates an output set that reflects 10 numbers [0,1,2,3,4,5,6,7,8,9] and represent the actual probability that the input depicts in regards to the 10 possible digits (see Figure 12).

Figure 12: Digit Classification

In a nutshell, the system basically transforms the original image into probabilities (the last update box in Figure 12 reflects the softmax - weight decay). During the process, the data represents many intermediate representations. Each hexagon depicted in Figure 12 represents a 28x28 reflection of the
numbers. It has to be pointed out that based on Figure 12, the network assigns only a 9% probability that the input actually is a number 5, but a whopping 60% probability that this number is an 8 (see output in Figure 12). This is not considered an issue at this point as by design, the mapping from input to output represents a mathematical function that is parameterized. During the training phase, the parameters are tuned so that over time, the network assigns a greater likelihood to class 5 for this particular input. Depending on the model being used, terms such as backpropagation or contrastive divergence are used to describe this process. Simplified, for this input, the probability of digit 5 is differentiated to derive the expression for the gradient in respect to all the other network parameters. The gradient reflects the vector in the parameter space along which the function (here the probability of depicting a 5). So the system forwards the image of the number 5, computes the probability that the answer is a 5, computes the gradient on all parameters of the network and finally (slightly) adjusts all the parameters along that gradient.

The amount by which the parameters are adjusted is labeled the learning rate and that rate probably reflects the single most important factor to consider while training these networks. If the learning rate is high, the networks learn faster but if the rate is too high, the networks can literally explode. To illustrate, too high of a learning rate makes the weights and objective function to diverge, so there is no learning at all. In contrast, if the learning rate is too low, the training cycle will take a very long time. In practice, the learning rate is normally calibrated to start high (for example 0.1) and is slowly annealed over time by a few orders of magnitude (as an example down to 0.0001). The train of thought is that after the adjustment, there is a better probability that the network predicts a 5 as the output. The network basically commences with some random parameters, loops through the procedure may times and so transforms itself into an actual digit classifier.

Summary & Some Additional Comments

With Deep Learning, the term dropout procedure is mentioned a lot. As the name suggests, during the training phase, a dropout procedure randomly drops hidden units (commonly with a binomial probability of 0.5). The goal is to harden the network in the sense that the network works around unnecessary unit co-activation scenarios. As a side effect, it is also a way to assure that the network does not just learn to compress the training set. Simplified, a dropout regularizer is first-order equivalent to an L2 regularizer that is applied after scaling the features by an estimate of the inverse diagonal Fisher information matrix. The Fisher information reflects a way to measure the amount of information that an observable random variable X carries about an unknown parameter \( \theta \) upon which the probability of X depends. A dropout procedure performs a gradient descent in respect to both, the training examples and the collection of all possible sub-networks, respectively. The regularization term represents the usual weight decay or Gaussian prior term based on the square of the weights to prevent overfitting. To reiterate, the learning rate represents a parameter that determines how much an updating step influences the current value of the weights. While the weight decay depicts an additional term in the weight update rule that causes the weights to exponentially decay to zero, if no other update is scheduled. A dropout procedure instantly provides the magnitude of the regularization term that is scaled by the inputs and by the variance of the dropout variables.

As already discussed, with a standard backpropagation procedure, if the learning rate is set too low, the network learns rather slowly. On the other hand, setting the learning rate too high results into a scenario where the weights and the objective function diverge (deviate) and hence there is no learning at all. If the objective function is quadratic (as with linear models) a decent learning rate can be derived via the Hessian matrix. On the other hand, if the objective function has many local and global optima (which is normally a characteristic of feedforward ANNs with hidden units), the optimal learning rate may change rather considerably during the training phase. Hence, trying to train an ANN by utilizing a constant learning rate is normally a daunting task that requires quite a bit of trial and error.
Many variations of backpropagation procedures have been developed over time. Most bear the same theoretical flaw as the standard backpropagation procedure. The crux of the issue is that the degree of the change in the weights (the step-size) should not be a function of the extent of the gradient. To illustrate, in some regions of the weight space, the gradient may be small and hence a large step size is needed (this scenario occurs when a network is initialized with small random weights). In other regions of the weight space, the gradient may be small and a small step size is needed (this scenario may occur when the system is close to a local minima). Likewise, a large gradient may require either a small or a large step. Many algorithms try to adapt the learning rate, but algorithms that multiply the learning rate by the gradient to compute the change in the weights is likely to produce a rather erratic behavior when the gradient suddenly changes. With some models, it is worth looking into a procedure such as Rprop. Rprop (resilient backpropagation) depicts a learning heuristic (a first-order optimization algorithm) for supervised learning in feedforward ANN's. Rprop only takes into account the sign of the partial derivative over all patterns (and not the magnitude) and acts independently on each weight. In general, conventional optimization algorithms not only use the gradient but also second-order derivatives or a line search (or some combination thereof) to attain a good step size. With incremental training procedures, it is much more difficult to devise an algorithm that automatically adjusts the learning rate during the training phase. While various proposals have been made, most of them do not provide an adequate solution to the problem. Some promising approaches are based on adapting the momentum rather than the learning rate while others utilize a variant of a stochastic approximation method that is labeled as iterate averaging that should theoretically provide optimal convergence rates by maintaining a running average of the weight values. But as the saying goes, your mileage may vary and this problem is most certainly still an open research challenge.

References