Overview of Deep Neural Networks

M. Kukačka
Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic.

Abstract. In recent years, new neural network models with deep architectures started to get more attention in the field of machine learning. These models contain larger number of layers (therefore “deep”) than conventional multi-layered perceptron, which usually uses only two or three functional layers of neurons. To overcome the difficulties of training such complex networks, new learning algorithms have been proposed for these models. The two most prominent deep models are the Deep Belief Networks and Stacked Auto-Encoders, which currently present state-of-the-art results in many traditional pattern recognition benchmarks. This article summarizes the advantages of deep architectures and provides a concise overview of the deep network models, their properties and learning algorithms.

Introduction

The multi-layered perceptron neural network has gained a reputation as a simple, trainable function approximator. The typical architecture for such a network consists of the input layer, where the input values are presented to the network, one hidden layer of neurons which process these input values, and an output layer of neurons, from which the output of the network can be read. This architecture is justified by numerous theoretical results, such as the well-known Kolmogorov’s theorem. These results show that any function (with reasonable constraints) can be represented in a form that can be directly translated to the described network of computational units.

As the field of machine learning begins to address more and more complex problems, the above described shallow network architecture seems to prove insufficient for solving them effectively. The above mentioned theoretical support for such an architecture fails to address the efficiency of representing functions in this form, much less the problem of efficient training of these networks. For complex functions, a shallow network can require very large numbers of computational units and equally large amounts of time for training. Problems such as over-fitting are also more likely to occur in this case.

A solution can be found in adding more hidden layers of neurons to the network, increasing the complexity of computation the network can perform, even with relatively small number of neurons. In theory, even complex functions can be represented concisely by such deep networks. However, increasing the number of hidden layers introduces new problems, particularly in training of the network. Standard algorithms for training of neural networks, based on gradient descent, typically perform poorly on deep networks, getting stuck in local optima and resulting in bad network performance and generalization. The cause is the principle of the gradient descent optimization. If the network is initialized with large weights, the network typically stays in the local area of parameter space, unable to shift the weights enough to move to an area of the weights space representing the optimal solution. If the network is initialized with small weights, these weights do not allow the information about the gradient of the error function, computed in the output layer, to sufficiently propagate into the lower layer and act on parameters of neurons in these layers. The resulting network then works as if the lower layers were not trained at all.

Several solutions were found for addressing this problem of training deep neural network architectures. One of the unique solutions, represented by the LeNet network model [LeCun et al., 1998], consists of constructing the network with special constraints (e.g. weight-sharing, limited neuron fan-in, asymmetric inter-layer connections), which lower the number of free parameters in the network and, at the same time, limit the network to smaller area of parameter space where a good solution is more likely to be found. The lower number of free parameters, combined with the weight-sharing technique, helps the gradient descent in acting strongly even in lower layer of the network. The architecture of the LeNet network is intended to aid the network in processing of image data, and with impressive results: for a long time, this model represented the best solution for numerous image processing tasks.

Another line of solutions for the deep training problem aims to initialize the lower layers of the network to an area of the parameter space where a good solution is likely to be found. For this task, unsupervised training of the lower layers is typically used, progressing from the first layer of the network...
and using outputs of already trained lower layer as inputs for training the subsequent one. In some cases, the whole network can then be fine-tuned by traditional gradient descent algorithms. Other solutions use the gradient descent algorithms to train only the top-most layers, using the lower layers as feature detectors or encoders, with the top layers performing standard classification of outputs of these detectors.

In the following chapters, two most prominent models of deep neural networks are described: Deep Belief Networks and Stacked Auto-Encoders. These models were proposed quite recently and already perform very well in many pattern recognition tasks and standard benchmarks.

**Stacked Auto-Encoders**

The Stacked Auto-Encoders (SAE) model, described in [Bengio, 2009], solves the obstacles of additional hidden layers by unsupervised initialization of the lower layers. Supervised criterion is then used to train the final layers of the network, and can be also used to fine-tune the parameters of the whole network.

The basic component of this model is the auto-encoder network. Its structure is shown in Figure 1a. This is a perceptron network with one hidden layer, trained to output the same values it gets on its input. The input and output layers therefore have the same number of units. The output of the hidden layer can be thought of as a code, into which the input is encoded by the hidden layer, and from which the output is again decoded by the output layer. In a trained network, these encoding and decoding operations are inverse to each other, in order to reconstruct the input values in the output layer.

Typically, the dimension of the code (i.e. the number of units in the hidden layer of the auto-encoder) is lower than the dimension of the input (as well as output) vector. In such a case, the encoding can be considered to be a compression of the input values. In order to allow good reconstruction of the input values from the code, this compression must be as lossless as possible. Because of the low dimension of the code, the encoding cannot be a good compression for the whole input space. The training therefore aims to create a good encoding only for the training set of values.

For training of an auto-encoder network, algorithms based on gradient descent, such as back-propagation, can be used. This allows all the traditional variants of these algorithms, such as training with momentum, to be used. Additional constraints can be enforced to improve generalization of the auto-encoder, e.g. sparsity of the code.

For construction of the SAE network, only the encoding stage of the trained auto-encoder network is used to create a single layer of the resulting network. The first auto-encoder is therefore trained to encode the values of the training set, the resulting codes are then used as input values for training of the subsequent auto-encoder, and so on. Such training of stacked encoding stages requires no information about labeling of the input samples. After this unsupervised stage of training of the SAE network is finished, an output layer of neurons is added, which is trained in a supervised way with inclusion of the labels (or target values) of the input samples. Using a supervised criterion (e.g. MSE error function), not only the final layer can be trained, but since the structure of stacked auto-encoders allows for back-propagating of the output error, the whole network can be fine-tuned to improve the overall performance.

![Figure 1. Structure of the basic components of deep networks: a) Auto-Encoder, b) Restricted Boltzmann Machine](image)

**Denoising Auto-Encoders**

The SAE model has been improved by using denoising auto-encoders instead of regular auto-encoders in the unsupervised stage of training. This improvement is described in [Vincent et al., 2008]. The denoising auto-encoders apply random corruption to the input values while still requiring the reconstruction to be the original, non-corrupted input values. The used form of corruption consists of
resetting part of the input values to zero, although other forms of corruption are possible (e.g. various types of noise, occlusions).

The denoising auto-encoders are forced to learn deeper correlations between the input values, and to reflect them in the resulting code to allow for good reconstruction. This results in more robust encoding of the input values, where the code reflects more substantial features of the training samples. Using these denoising auto-encoders, the Stacked Denoising Auto-Encoders model is formed, which gives better results in pattern recognition tasks than the regular SAE model.

Deep Belief Networks

The Deep Belief Network (DBN) model was proposed in [Hinton, 2006a]. Much like the Stacked Auto-Encoder model described in the previous chapter, the DBN network is composed of stacked modules, which can be trained in unsupervised manner, and can be complemented by final top stage for classification. In the case of DBNs, the individual modules take the form of Restricted Boltzmann Machines (RBMs), a two-layered neural networks with connections only between the layers. The Restricted Boltzmann Machine network, its function and learning algorithm are described in more detail in the following sub-section.

Similarly to the training of SAE network, the DBN model is trained one module after another, proceeding from the lowest RBM and using its outputs (after its training is finished) as inputs for training the subsequent RBM module. Each RBM is trained in unsupervised manner. After the training of all RBM modules is finished, the output of the final one can be used as representation of the inputs with reduced dimensionality (see [Hinton, 2006b]), or as input into classification layer. Alternatively, an auto-associative memory network can be trained as the last module, with label information added to the top layer, to add classification capability to the network. Also, the inner functionality of RBM modules allows their parameters to be used for initialization of deep perceptron network, which can then be fine-tuned by supervised criterion (e.g. back-propagation algorithm).

Restricted Boltzmann Machines

A Restricted Boltzmann Machine (RBM) is a two-layered neural network with connections only between the two layers of neurons. Its structure is demonstrated in Figure 1b. It is a variant of Boltzmann Machine model, in which any two neurons can be connected. Such unconstrained connectivity turned out to be a disadvantage - it made learning algorithms inefficient. For the restricted variant, on the other hand, an efficient algorithm exists and will be described below.

An RBM network is composed of two layers of neurons. The lower one is called visible (or input) layer, and the output values of its neurons will be denoted as \( v \). The higher layer is called hidden, and outputs of its neurons will be denoted \( h \). The weight of connection between neurons \( v_i \) and \( h_j \) is denoted by \( w_{ij} \), and all connections are symmetric, meaning that \( w_{ij} = w_{ji} \). The visible and hidden neurons also have bias parameters, denoted as \( b_i \) and \( c_j \), respectively.

The RBM belongs to the family of energy-based models, along with e.g. the Hopfield network. The global energy function of an RBM network, given the values of visible neurons \( v \), hidden neurons \( h \), vectors of biases \( b \) and \( c \), and the matrix of connections weights \( W \), is described by the following equation:

\[
\text{Energy}(v, h) = -b^Tv - c^Th - h^Twv \tag{1}
\]

The neurons in RBM are binary and stochastic, meaning that each neuron outputs values 0 or 1 with certain probability, which is dependent on the values of neurons it is connected to. From the energy equation, the probability of a neuron outputting the value 1 can be derived (see [Bengio, 2009] for more details). The resulting probability function is following:

\[
P(h_j = 1|v) = \frac{1}{1 + \exp(-c_j - \sum_i v_i w_{ij})} = \logsig(c_j + \sum_i v_i w_{ij}) \tag{2}
\]

Since the visible and hidden neurons have symmetric roles in the energy function, the probability function of \( P(v_i|h) \) is analogous.

The network performs computation in the following way: first, the values of the visible units \( v \) are fixed to the input values \( x \). In the next step, values of the hidden units \( h \) are sampled from their respective distributions \( P(h_j = 1|v) \). In general Boltzmann Machine, the unit to be updated would be chosen randomly, but since the RBM has no intra-layer connections, we can update all hidden units at
once. This is the so-called positive phase, while negative phase of the machine’s calculation consists of sampling values of the visible units from their respective distributions $P(v_i = 1|h)$. The negative phase is instrumental in training of the network. When using the RBM as a part of a DBN network’s processing of input values, only the positive phase is needed. In that case, it can be used in several different ways, such as using the actual probabilities $P(h_j = 1|v)$ as output values of the hidden neurons instead of binary values sampled from these distributions. See [Hinton, 2010] for more information on using RBMs as modules in DBN networks.

**Contrastive Divergence**

The Contrastive Divergence (CD) algorithm has been proposed by Hinton in [Hinton, 2002] for efficient training of RBM networks. The aim of the algorithm is to approximate log-likelihood gradient descent in order to model the distribution of the input values by the distribution learned by the RBM as closely as possible. The algorithm is described by pseudo-code in Algorithm 1.

The algorithm consists of performing Gibbs sampling of values from the network, i.e. of alternating the positive and negative phases of network’s computation. In the first step, the values of the visible units $v$ are fixed to the input values $x = v_1$. Then the values of the hidden units are sampled in the first positive phase: $h_1 \sim P(h|v_1)$. From these values $h_1$, the reconstruction of values of the visible units is sampled: $v_2 \sim P(v|h_1)$, and so on for $h_2$, etc. Variations of the CD algorithm differ in the number of these steps, e.g. for CD-1 the sampling ends with sampling of $h_2$. In order to get higher versions of the algorithm (CD-2, CD-3, etc.), the steps 2 through 9 of Algorithm 1 can be repeated multiple times.

After the appropriate number of Gibbs sampling steps is performed, the weights and biases of the network are adjusted. The adjustments are based on the difference between the first values in the chain ($v_1$, $h_1$) and the last values ($v_2$, $h_2$ in case of CD-1). An interesting property of this algorithm is that the adjustment of a weight $w_{ij}$ depends only on the activations of units $v_i$ and $h_j$, which is biologically plausible. Allegedly, even CD-1 is sufficient for successful training, even though CD-k for higher values of $k$ produce better approximation of the likelihood gradient. More detailed description of the CD-1 algorithm can also be found in Section 5.4 of [Bengio, 2009], for full theoretical justification of the algorithm see [Hinton, 2002].

**Algorithm 1** Contrastive Divergence (CD-1) algorithm

$x$ is the input vector

$v, h$ are the vectors of output values of visible units and hidden units, respectively

$w_{ij}$ is weight between visible unit $i$ and hidden unit $j$

$\epsilon$ is the learning rate

$b_i, c_j$ are biases of input unit $i$ and hidden unit $j$, respectively

1: $v_1 \leftarrow x$
2: **for all** hidden units $j$ **do**
3: \hspace{1em} compute probability $P(h_{1j} = 1|v_1)$ $\triangleright$ typically: $\log\text{sig}(c_j + \sum_i w_{ij}v_i)$
4: \hspace{1em} sample $h_{1j} \in \{0, 1\}$ from $P(h_{1j}|v_1)$
5: **end for**
6: **for all** visible units $i$ **do**
7: \hspace{1em} compute probability $P(v_{2i} = 1|h_1)$ $\triangleright$ typically: $\log\text{sig}(b_i + \sum_j w_{ij}h_j)$
8: \hspace{1em} sample $v_{2i} \in \{0, 1\}$ from $P(v_{2i}|h_1)$
9: **end for**
10: **for all** hidden units $j$ **do**
11: \hspace{1em} compute probability $P(h_{2j} = 1|v_2)$ $\triangleright$ typically: $\log\text{sig}(c_j + \sum_i w_{ij}v_i)$
12: **end for**
13: **for all** weight values $i, j$ **do**
14: \hspace{1em} $w_{ij} \leftarrow w_{ij} + \epsilon(v_{1i}h_{1j} - v_{2i}P(h_{2j} = 1|v_2))$
15: **end for**
16: $b \leftarrow b + \epsilon(v_1 - v_2)$
17: $c \leftarrow c + \epsilon(h_1 - P(h_{2j} = 1|v_2))$

Generative properties of DBN model

An interesting property of the DBN model is that it is generative, in contrast to the discriminative nature of the traditional neural networks. This means that not only is the network able to learn and
sample from the distribution $P(\text{Label}|\text{Data})$, but after it is trained, it is also able to sample from the distribution $P(\text{Data}|\text{Label})$. This requires some adjustments to the procedure of training the final stage of the network and fine-tuning the network. However, a network trained in this way can then be questioned about representatives of the classes it has learned. These representatives are created by an top-down pass of the network, starting from the label information in the top layer and propagating them downward using the $P(v|h)$ distribution in each RBM. This results in stochastic reconstruction of a sample from the indicated class. In [Hinton, 2006a], the authors liken this to “observing the mental state” of the network. In this article, examples of such reconstructed samples can be found.

Convolutional DBNs

A combination of successful approaches has the potential of yielding better results. From the combination of the DBN model, with its generative nature and fast unsupervised training of lower layers, and the network structure of the LeNet model, suitable for processing images, comes the Convolutional DBN (CDBN) model. Proposed in [Lee et al., 2009], this model tries to combine the advantages of its sources of inspiration.

The combination of DBN and LeNet models is in many ways quite straightforward. The layers of the network are formed by Convolutional RBM (CRBM) modules, which are trained on small areas of the underlying layer (or the input image) in order to detect local features. The neurons are organized into feature maps, in which the neurons are constrained to share the same parameters (so-called weight-sharing), thus detecting the same features in all positions of the image. In this way, simple local features are detected in the lower layers (such as edges and lines of various orientation), and these simple features are combined into more complex ones in higher layers.

The only complication of using the LeNet network structure while keeping the generative property of DBNs is the application of the sub-sampling operation. This operation in the LeNet model is performed by sub-sampling layers, which alternate with the feature-detecting layers, and works by averaging small areas (such as 2x2) of individual feature maps. This hides the exact positions of features, making the network more robust with respect to translation. Also, the sub-sampling operation shrinks the feature maps, which helps to keep the network small and its operation efficient.

The nature of the sub-sampling operation is purely discriminative, the information about exact position of the features is being destroyed, therefore the sub-sampling layers allow only bottom-up signal flow. To keep the CDBN network generative, sub-sampling has been replaced by max-pooling operation, introduced in [Lee et al., 2009]. This takes the form of specialized CRBM module with further constraints on the activity of the underlying layer neurons.

There are several advantages in combining the properties of LeNet and DBN models. By using local feature detectors and weight-sharing, the CDBN network is easily scalable to large images while keeping the number of its parameters low. The generative nature of the network makes it possible to find out what features has the network extracted from the training data, and which features are used in processing of individual samples. With top-down flow of signal, it is possible to restore even substantially occluded or damaged samples, supplementing the missing simpler features by completing the more complex ones detected in higher layers.

Summary

Despite being introduced relatively recently, the deep neural network models, especially the Deep Belief Networks and Stacked (Denoising) Auto-Encoders, are gaining a well-deserved attention. For some problems and benchmarks, these models present the best achieved results - for comparison of results of different models, see for example [Vincent et al., 2008]. The fast unsupervised training algorithms, together with their unique properties (such as the generative property of the DBN networks) and impressive results on many standard detection and classification benchmarks, make these models very interesting for the area of pattern recognition.

Acknowledgments. The presented work was supported by the Czech Grant Agency under the contract no. 201/09/H057 and by SVV project number 265 314.

References