A CONTEXT BASED IMAGE RETRIEVAL SYSTEM USING AUTO-ENCODER AND RBM ON THE AMD PLATFORMS

Junli Gu, Maohua Zhu, Xuwen Tian

gujunli@gmail.com

ABSTRACT

This report briefly summarizes a GPU-acceleration project with the aim of providing an OpenCL implementation for auto-encoder and Restricted Boltzmann Machine (RBM), two state-of-the-art deep learning algorithms, on the AMD computing platforms, and the aim of trying to expand the potential of these platforms for Big Data related applications. In this report, a general description of these algorithms is provided to explain their implementation. As literature study, it also briefly describes a few fundamental concepts in machine learning and artificial intelligence, e.g., supervised training and unsupervised training. Thereafter, the advantages of using GPUs in the implementation are analyzed along with optimization strategies to fully exploit the computing power of GPU devices and OpenCL. This report provides deep analysis and guidance of how auto-encoder and RBM can be used, the application scenarios and their advantages. Finally, as proof point, we build a context-based image retrieval demo on top of the above two models to demonstrate the capability of unsupervised learning and the performance boost GPU devices provide.

1. BACKGROUND

In modern information systems, the sheer amount of the stored data, as well as their complexity, is increasing at a high rate. This trend is generally known as Big Data in industry or data-intensive (or analysis-driven) science in academia. To search and analyze the large amounts of complex objects, data mining and complex search (e.g. similarity search) offer new possibilities to maximize the utilization of available information.

Traditionally, statistics tools and machine learning tools are used to explore the relationship in the data. Statistics has been focused on data modeling—on the estimation of a probability law that has generated the data. Hence, standard statistical analysis, typified by regression, estimation, and hypothesis testing techniques, assesses parameters of a distribution from samples drawn of that distribution. With the help of such parameters, people can infer associations among variables, estimate the likelihood of past and new events, as well as update the likelihood of events in light of new evidence or measurements. These analysis tasks are managed well by standard statistical analysis so long as experimental conditions remain the same. Besides, machine learning and artificial intelligence tools are widely used for their excellent performance in finding useful information in the data. Machine learning has been focused on prediction: given observations that have been generated by an unknown stochastic dependency, the goal is to infer a law that will be able to correctly predict future observations generated by the same dependency. Precisely speaking, machine learning is first to learn from data the structure and parameters of an optimal predictive model. This task is referred to as model inference. Once a model structure and its parameters are learnt, another kind of inference can take place: the inference of variable statistics (point estimations, estimation of expectations, or distribution calculation) given the values of other variables. This task can be referred to as variable inference to distinguish from the first one.

In fact, the astonishing exponential growth and wide availability of digital data make it not an ordinary task to harvest valuable knowledge from Big Data. The sheer size of data available presents unprecedented challenges to harnessing data and makes it difficult or even impossible to manage and analyze using conventional tools and technologies. Now, deep learning for the artificial neural networks (ANN) with deep architecture, together with advances in available computational power, have come to play a vital role in Big Data analytics and information/pattern extraction. ANN

models are important tools commonly used in machine learning and artificial intelligence applications. Conventionally, most ANN models used a few types of shallow-structured learning architecture. By contrast, deep learning refers to the deep architecture of using supervised or unsupervised offline learning strategies to automatically learn hierarchical feature representations for later online prediction. Deep learning, inspired by biological observations on human brain mechanisms for processing of natural signals, has attracted much attention from the academic community in recent years, due to its state-of-the-art performance in many research domains, e.g., speech recognition and computer vision.

The networks with deep architecture have shown significant promises as a technique for learning features for automatic recognition systems. These networks typically consist of multiple layers of simple computational elements. By combining the output of lower layers in higher layers, these networks can represent progressively more complex features of the input. Hinton et al. introduced deep belief network (DBN), in which each layer consists of a Restricted Boltzmann Machine. Bengio et al. built the deep network using an auto-encoder neural network in each layer. Ranzato et al. and Lee et al. explored the use of sparsity regularization in auto-encoding energy-based models and sparse convolutional DBN models with probabilistic max-pooling, respectively. It has been reported that these networks, when trained subsequently in a discriminative fashion, can achieve excellent performance on handwritten digit recognition tasks. Furthermore, Lee et al. and Raina et al. show that deep networks are able to learn good features for classification tasks even when trained on data that does not include examples of the classes to be recognized.

Deep neural networks (DNN) made outstanding success in some applications but encountered challenging problems too. Some problems can be tackled by using computing platforms with heterogeneous architecture (e.g., those integrated with the GPU or ASIC accelerators) for massive computation tasks. For example, DNN has become a popular acoustic modeling, showing significant gains over Gaussian Mixture Model/Hidden Markov Model (GMM/HMM) systems on vocabulary tasks of different sizes. The development of pre-training algorithms and better forms of random initialization, as well as the availability of faster computers, has made it more achievable to train deep networks than before. It has been shown that the performance of DNN improves with increasing training data, and in practice the deep networks have achieved excellent performance. Training DNN models however remains to be markedly slow, for instance, for large vocabulary continuous speech recognition tasks. This issue is mainly due to several reasons: (1) the models for real-world speech tasks are trained on hundreds of hours of voice data, which amounts to millions of training examples; (2) roughly 10–50 million network parameters are used for speech tasks, which is much larger compared to the number of parameters used with those common acoustic modeling approaches (e.g. GMM) on the same tasks; and (3) the popular methodology to train deep networks still uses stochastic gradient descent, serially in nature on one machine. Hence, the GPU devices can be used in some research projects to address intensive computation problems.

This report is organized as follows. First, a general description is made for the explanation of using DNN models or deep learning in Big Data applications. Then, a brief description is made for the key concepts of supervised learning and unsupervised learning as well as their usage in the training of neural networks. Second, the reasons for using the GPU devices are presented. Third, auto-encoder and RBM, i.e. two state-of-the-art unsupervised models, are described. Fourth, the reasons for their OpenCL implementation using GPU devices are presented along with primary optimization strategies of fully using the computing power of the GPU. Last, a real-world application is presented for illustration.

2. REASONS FOR USING DEEP LEARNING ALGORITHMS

This section first gives a general description of ANN models and then three primary reasons for using ANN in the big data applications with respect to three main characteristics of these applications, respectively.

2.1 NEURAL NETWORK

An ANN model can be viewed as a massively parallel computing system, which consists of an extremely large number of simple processors with many interconnections. ANN models attempt to use some organizational principles (e.g. learning, generalization, adaptivity, fault tolerance and distributed representation, and computation) in a network of weighted directed graphs in which the nodes are artificial neurons and directed edges (with weights) are connections between neuron outputs and neuron inputs. The main characteristics of neural networks are that they have the ability to learn complex nonlinear input-output mapping relationships, use sequential training procedures, and adapt themselves to the data.

The most commonly used family of ANN models for pattern classification tasks is the feed-forward network, including multilayer perceptron (MLP) and Radial-Basis Function network. These networks are organized into layers and have unidirectional connections between the layers. Another popular network is Self-Organizing Map (or Kohonen-Network), which is mainly used for data clustering and feature mapping. The learning process involves updating network architecture and connection weights so that a network can efficiently perform a specific classification/clustering task. The increasing popularity of neural network models to solve pattern recognition problems is primarily due to their seemingly low dependence on domain-specific knowledge (relative to model-based and rule-based approaches) and due to the availability of efficient learning algorithms for practitioners to use.

ANN models provide a suite of nonlinear algorithms for feature extraction (using hidden layers) and classification. In addition, existing feature extraction and classification algorithms can be mapped on neural network architectures for efficient (hardware) implementation. In spite of the seemingly different underlying principles, most of the well-known neural network models are implicitly equivalent or similar to classical statistical pattern recognition methods. Discussions have been made on this relationship between neural networks and statistical pattern recognition. It was pointed out that “neural networks are statistics for amateurs... Most ANNs conceal the statistics from the user.” Despite these similarities, neural networks offer several advantages, including unified approaches for feature extraction and classification and flexible procedures for finding good, moderately nonlinear solutions.

ANN models (e.g. MLP) are also able to handle high dimension problems, whatever the dimension of the signal is. High dimensional output spaces may be verbose and are likely to contain internal dependencies, especially if the task is not well understood or badly formulated. In the past, these models were studied to approximate non-linear functions in many decision tasks. For example, the standard MLP relies on the perceptron principle and its optimization is based on the back-propagation algorithm. In fact, the success of learning tasks usually requires the manual engineering work as well as a large amount of labeled data. This preparation work often is expensive. For example, representation features are hand-designed, and their design requires significant amounts of domain knowledge and human labor, and does not generalize well to new domains.

In a multi-layer network, the activation state values of each layer of neurons can be seen as a representation of the input data. Learning algorithms that learn to represent the raw data with multiple levels of composition are said to have the deep architecture, which generally has three or more hidden layers, compared with those previous shallow models with only one or two hidden layers. It is conjectured that the deep architecture is much more powerful in feature representation (much larger parameter space with deeper network and wider layer size) than those shallow ones. The central idea of the deep learning design relates to the concept of stacking, and simple modules of functions or classifiers are composed first and then they are “stacked” on top of each other in order to learn complex functions or classifiers. From the mathematical perspective, each layer represents a non-linear mapping relationship; stacked structure thus enables a very powerful multiple-layer non-linear mapping space from input data to final output. From the signal processing perspective, deep networks consist of many-layer feature detectors, lower layers detect simple features and feed into higher layers, which in turn detect more complex/advanced features. Lay by layer, the deep network will

extract the so-called hierarchical feature learning, which explains why deep learning is powerful. The learning of deep architecture focuses on automatically discovering/extracting the abstractions from the low-level features to the high-level concepts with as little human effort as possible, i.e., without having to manually define all necessary abstractions or having to provide a huge set of relevant hand-labeled examples. These learning algorithms would certainly help to transfer much of human knowledge into machine-interpretable form. The deep learning process is said to resemble how human brains or human eyes digest captured information step by step and finally form final classification/understanding.

Deep learning systems are inspired by the human brain, which appears to process the information through multiple stages of transformation and representation, especially in the primate visual system. The sequence of processing stages includes the detection of edges, primitive shapes, and gradually moving up to complex shapes. Low-level abstractions are more directly tied to particular percepts, whereas high-level ones are called "more abstract" because their connection to actual percepts is more remote, and through other intermediate-level abstractions. Each level of abstraction found in the brain consists of the "activation" (neural excitation) of a small subset of a large number of features that are, in general, not mutually exclusive. These features are not mutually exclusive, so they form what is called a distributed representation: the information is not localized in a particular neuron but distributed across many. In addition to being distributed, it appears that the brain uses a sparse representation—only around 1-4% of the neurons are active together at a given time. In fact, the real-world data carry rich structures, even though the data are unlabeled. Given natural images, people are able to discover low-level structures (e.g. edges) and high-level structures (e.g. corners, local curvatures, and shapes). Generally, high-level feature detectors need information from progressively larger input regions. These structures in different scales can be useful in object recognition/classification tasks, which is a main assumption of unsupervised feature learning. Deep learning has been recognized as an efficient way of finding representations in different scales from the unlabeled data.

The research on deep learning aims at developing methods which learn feature hierarchies with high-level features of the hierarchy formed by the composition of low-level features. Automatically learning features at multi-levels of abstraction allow a system to learn complex functions mapping the input to the output directly from data, without depending heavily on manually crafted features. This is especially important for high-level abstractions, which humans often do not know how to specify explicitly in terms of raw sensory input. The ability to automatically learn powerful features will become increasingly important as the amount of data and range of applications to machine learning methods continues to grow. However, conventional approaches to train ANN models are limited to a few hidden layers. The weights of the ANN training are initialized randomly and the objective function is non-convex, so the weights tend to get stuck in a poor local optimum. The concept of pre-training was put forward as a solution for the network training. The main idea to pre-train ANN models is to provide a good initialization of the weights and to include regularization in the training. The methods include the Restricted Boltzmann Machine (RBM) and auto-encoder.

2.2 DEEP LEARNING For BIG DATA

Leveraging deep learning to extract valuable patterns and information from Big Data has been the industry's major goal and targets. For example Baidu has been using deep learning to discover the population migration trend during holidays. Alibaba has been using machine learning for personalized recommendation on their electronic business. However Big Data plus deep learning has exposed great challenges with regards to DNN model design, training and underlying hardware platforms. Big data possesses a large number of training samples, large varieties of class types, and very high dimensionality. To give a quantitative understanding, the recent scale of Big Data in industry includes millions for image classification, billions for voice recognition. Yet projected data tends to grow 10x per year. These properties directly lead to running-time complexity and model complexity, which makes it impossible to train a deep learning algorithm with a central processor and storage. Instead, distributed frameworks with parallelized machines are the enablers for Big Data

applications and thus have been investigated in past two years. . Impressive progresses have been made to mitigate the challenges related to high volumes.

Fortunately Big Data + DNN implies massive level of data parallelism and intrinsic model parallelism. In literature people believed that such properties match well with GPU's many-thread parallelism and GPU enabled the success of deep learning. The most recent deep learning frameworks use clusters of CPUs or GPUs to decrease the training time while still guarantee algorithm convergence, because industry cares most about the time to get the results (eg recognition/classification results). Internet giants, eg Google and Baidu , build DNN models with millions to billions of parameters to train Big Data application. The continuous growth in computer memory and computational power (mainly through parallel or distributed computing environment) helps to build the deep learning systems needed for scaling-up to very large data sets. Besides those common issues associated with computation and communication management (e.g., copying data or parameters or gradient values to different machines. These high performance computing infrastructure-based systems also employ data parallelism or model parallelism or both. For example, the data batch is divided into blocks called mini-batch, models are divided into blocks called model replica; the forward and backward propagations can be implemented effectively in pipeline style, although these algorithms are not trivially parallel.

Big Data sets are collected through internets such as search engines or social networking interfaces. Thus it brings another challenge: data unlabled with noise. The availability of unlabeled Big Data presents ample opportunities for internet companies to extract real life patterns and information, meanwhile it also poses great challenges to train such unlabeled noisy data. Strategies used in unsupervised learning and semi-supervised learning may also help alleviate problems related to noisy labels.

Unsupervised learning has the ability to utility unlabeled data during training: learning data distribution without using label information. Advanced deep learning methods are required to deal with noisy data and to be able to tolerate some messiness.

2.3 DEEP LEARNING FOR HIGH VARIETY OF DATA

The second advantageous aspect of deep learning is its ability for representation learning - with either supervised or unsupervised methods or combination of both, deep learning can be used to learn good feature representations for classification. The data comes in all types of formats from a variety sources, probably with different distributions. For example, the rapidly growing multimedia data coming from the Web and mobile devices include a huge collection of still images, video and audio streams, graphics and animations, and unstructured text, each with different characteristics. DNN models can handle the data in an integrated fashion, which is a key to deal with the high variety of the data. These models are able to discover intermediate or abstract representations, which is carried out using unsupervised learning in a hierarchy fashion: one level at a time and higher-level features defined by lower-level features. Thus, a natural solution to address the data integration problem is to learn data representations from each individual data sources using deep learning methods, and then to integrate the learned features at different levels.

It has been that deep learning is very effective in integrating data from different sources. For example, Ngiam et al. developed a novel application of deep learning algorithms to learn representations by integrating audio and video data. They demonstrated that deep learning is generally effective in (1) learning single modality representations through multiple modalities with unlabeled data and (2) learning shared representations capable of capturing correlations across multiple modalities. Most recently, Srivastava and Salakhutdinov developed a multimodal Deep Boltzmann Machine that fuses two very different data modalities, real-valued dense image data and text data with sparse word frequencies, together to learn a unified representation. DBM is a generative model without fine-tuning: it first builds multiple

stacked-RBMs for each modality; to form a multimodal DBM, an additional layer of binary hidden units is added on top of these RBMs for joint representation. It learns a joint distribution in the multimodal input space, which allows for learning even with missing modalities. Because current experiments have demonstrated that deep learning is able to utilize heterogeneous sources for significant gains in system performance, deep learning seems well suited to the integration of heterogeneous data with multiple modalities due to its capability of learning abstract representations and the underlying factors of data variation. With the computing power supported by the GPU devices, the above expectation cannot come into being in the real-world applications.

2.4 DEEP LEARNING FOR HIGH VELOCITY OF DATA

Emerging challenges for Big Data learning arose from high velocity: data are generating at extremely high speed and need to be processed in a timely manner. One solution for learning from such high velocity data is online learning approaches. Online learning learns one instance at a time and the true label of each instance will be available quickly, which can be used for refining the model. This sequential learning strategy particularly works for Big Data as current machines cannot hold the entire dataset in memory. While conventional neural networks have been explored for online learning, only limited progress on online deep learning has been made in recent years. Interestingly, deep learning is often trained with stochastic gradient descent approach, where one training example with the known label is used at a time to update the model parameters. This strategy may be adapted for online learning as well. To speed up learning, instead of proceeding sequentially one example at a time, the updates can be performed on a mini-batch basis. Practically, examples in each mini-batch are as independent as possible. Mini-batches provide a good balance between computer memory and running time.

Another challenging problem associated with the high velocity is that data distribution is non-stationary, i.e., changing over time. Practically, non-stationary data are normally separated into chunks with data from a small time interval. The assumption is that data close in time are piece-wise stationary and may be characterized by a significant degree of correlation and, therefore, follow the same distribution. Thus, an important feature of a deep learning algorithm is the ability to learn the data as a stream. One area that needs to be explored is deep online learning - online learning often scales naturally and is memory bounded, readily parallelizable, and theoretically guaranteed. Algorithms capable of learning from non I.I.D. data are crucial for Big Data learning.

Deep learning can leverage both high variety and velocity of Big Data by transfer learning or domain adaption, where training and test data may be sampled from different distributions. Glorot et al. implemented a stacked de-noising auto-encoder based deep architecture for domain adaption, where one trains an unsupervised representation on a large number of unlabeled data from a set of domains, then applied this encoder to train a classifier with few labeled examples from only one domain. Their empirical results demonstrated that deep learning is able to extract a meaningful, high-level representation that is shared across different domains. The intermediate high-level abstraction is general enough to uncover the underlying factors of domain variations, which is transferable across domains. Bengio et al. applied deep learning of multiple level representations for transfer learning where training examples may not well represent test data. They showed that more abstract features discovered by deep learning approaches are most likely generic between training and test data. Deep learning therefore is a top candidate for transfer learning because of its ability to identify shared factors present in the input. Although preliminary experiments have shown much potential of deep learning in transfer learning, the application of deep learning to this field is relatively new and much more needs to be done for improvement. The big question is still if people can benefit from Big Data with DNN for transfer learning.

In conclusion, deep learning models are attractive for their superior performance in modeling high-dimensional richly structured data. A deep learning model learns multiple levels of representation and abstraction that help to make sense

of data. Big Data equally presents significant challenges to deep learning, including large scale, heterogeneity, noisy labels, and non-stationary distribution, among many others. To realize the full potential of Big Data, we need to address these technical challenges with new ways of thinking and transformative solutions. It has been asserted that these challenges posed by Big Data are not only timely, but will also bring ample opportunities for deep learning. Together, they will provide major advances in science, medicine, and business.

3. SUPERVISED LEARNING AND UNSUPTERVISED LEARNING

In a deep network, each layer of neurons can be seen as a representation of the input. This internal representation is obtained through a learned transformation. A good representation should disentangle the factors of variation which inherently explain the structure of the distribution. Such a representation may preserve the information about the input while being easier to model than the input itself. When a representation is used in a prediction/classification task, it may be such that there exists a "simple" (i.e., somehow easy to learn) mapping from the representation to a good prediction. To constructively build such a representation, it has been proposed to use a supervised criterion at each stage. However, the use of a supervised criterion at each stage may be too greedy and does not yield as good generalization as using an unsupervised criterion. Aspects of the input may be ignored in a representation tuned to be immediately useful (with a linear classifier) but these aspects might turn out to be important when more layers are available. Combining unsupervised (e.g., learning about p(x)) and supervised components (e.g., learning about p(y|x)) can be helpful when both functions p(x) and p(y|x) share some structure. For the sake of integrity, this section gives a brief description of supervised learning and unsupervised learning.

3.1 SUPERVISD LEARNING

In machine learning, supervised learning is the problem of learning input-output mappings from the empirical data, i.e., the training dataset. Depending on the characteristics of the output, this problem is known as either regression, for continuous outputs, or classification, when outputs are discrete. In supervised learning, training is accomplished by analyzing a set of provided, pre-classified examples to determine how to appropriately modify the connection weights so that these training examples are more accurately classified.

3.2 UNSUPERVISED LEARNING

Unsupervised learning is closely connected to the topics of regularization and compression. Unsupervised learning is normally used to encode raw incoming data such as video or speech streams in a form that is more convenient for subsequent goal-directed learning. Unsupervised learning algorithms mainly aim at discovering the structure hidden in the input data, capturing the regularities in the raw data for the purpose of extracting useful representations or restoring the corrupted data, and learning good representations that describe the raw data in a less redundant or more compact way can be fed into supervised learning machines, whose search spaces may thus become smaller than those necessary for dealing with the raw data. Good representations usually eliminate irrelevant kinds of variability of the input data, while preserving the information that is useful for the ultimate task. These algorithms can build only roughly-correct explanations. These explanations need not be proofs, and rather they need be reasonable arguments as to why the current item belongs to a specific category or why the current plan meets the given plan.

Unsupervised learning primarily uses unlabeled data, primarily because of the assumption that there exists rich intrinsic structures/patterns in the unlabeled data, even though the data are not labeled with categories. These structures are useful in machine learning tasks. Specifically, if some structures are generated from specific object classes, e.g. cars or faces, then these discovered class-specific patterns (e.g., car wheels or face parts) are likely to be useful for classification, possibly combined with a small amount of labeled data. In many cases, it is desirable for a learning algorithm to

distinguish semantically distinct patterns. The performance of an object recognition algorithm may be improved if it can separate foreground object patterns from background clutters. However, it is still challenging to learn useful high-level features from the data which contains a significant amount of irrelevant patterns.

Unsupervised learning algorithms appear diverse due to their dissimilar principles, but most of these methods use a common viewpoint. These algorithms use a scalar-valued energy function E(Y) that operates on the input data vectors Y. The energy function is designed to produce low energy values when Y is similar to some training data vectors and high energy values when Y is dissimilar to any training data vector. Thus, training an unsupervised model consists in searching for an energy function within a family {E(Y, w), w$\in$***W***} indexed by a parameter w that gives low energy values on the input data similar to the training samples and large values on those dissimilar ones. From the standpoint of generative models with latent variables, unsupervised learning algorithms learn salient structure patterns from the input data. In this view, the posterior probabilities of the latent variables can be used as features for discriminative tasks. Some unsupervised methods are based on reconstructing the input from the representation, while constraining the representation to have certain desirable properties (e.g. low dimension and sparsity). Others are based on approximating density by stochastically reconstructing the input from the representation.

Deep neural networks are black boxes that are difficult to train and understand. These days industry still focuses on supervised learning and gets signification improvement in accuracy, eg for image classification and voice recognition. Meanwhile unsupervised learning on top of unlabeled data is the coming challenge. Google and Baidu has gathered great talents and claimed efforts in this direction. In 2014 May, Baidu hired machine learning expert Andrew Ng who claimed goal of conquering unsupervised learning using Baidu's Big Data sets. The problem of training deep networks was recently focused on unsupervised learning. This emphasis is crucial for industry because there is usually a lot more unlabeled data compared with labeled ones. The solution considers each layer as an unsupervised generative gradient-based module that learns from its input distribution and stacks them one layer at a time from the bottom-up, in a greedy manner. This makes it scale well to large-sized networks. It also appears sensible to learn simple representations first and higher level abstractions on top of existing lower-level ones. In place of randomly initialized parameters, this unsupervised representation forms the initialization—a catalyst to learn meaningful representations—for the subsequent supervised discriminative learning phase. Towards today, to our knowledge there are three popular methods to learning a network of fully connected layers, namely: DBN, deep auto-encoder, and deep sparse coding.

4 NEURAL NETWORKS FOR UNSUPERVISED LEARNING

This section describes briefly three important neural networks for unsupervised learning—Boltzmann Machine (BM), Restricted Boltzmann Machine (RBM), and auto-encoder. These models are non-linear feature extraction methods, and there is no label information involved in the procedure of training these models, because these models characterize the input data distribution using hidden variables. The features extracted aim at conserving and better representing information, instead of performing classification tasks, although these goals are correlated sometimes. However, when the label information is available, it can be used together with the data to form the concatenated "dataset".

The auto-encoder is a special type of the mirrored feed-forward network, as shown in Fig 3., whose left mirror part encodes the input data into narrowed codes (as representations in feature space) and right mirror part reconstructs the input data from codes. Autoencoder is unsupervised learning in that it requires no label of input data, instead the reconstruction error between original input and recnstructed output is used to train the network. So the training goal of autoencoder is to be able to reconstruct the original data with very little error. Its effectiveness is to learn a representation or to encode of the original data, in the form of input vectors, at hidden layers. A BM is a parameterized model representing a probability distribution, and it can be used to learn important aspects of an unknown target

distribution based on samples from this target distribution. Training a BM means adjusting its parameters such that the probability distribution the machine represents fits the training data as well as possible. In general, the computation required to learning a BM model is demanding. RBM is a special case of the generic BM model and it is a successful generative model. RBM by itself is limited in what it can represent, because it is a bi-directionally connected network of stochastic processing unit, but its real power emerges when multiple RBM models are stacked to form a deep belief network (DBN)—a generative model consisting of many layers. In addition, deep networks, including MLP and DBN, can be trained by greedily training each hidden layer from the lowest layer to the highest layer as an RBM using the previous layer's activation results as input. This greedy layer-wise training approach has been shown to provide a good initialization for parameters for the deep architecture, as shown in fig 1.

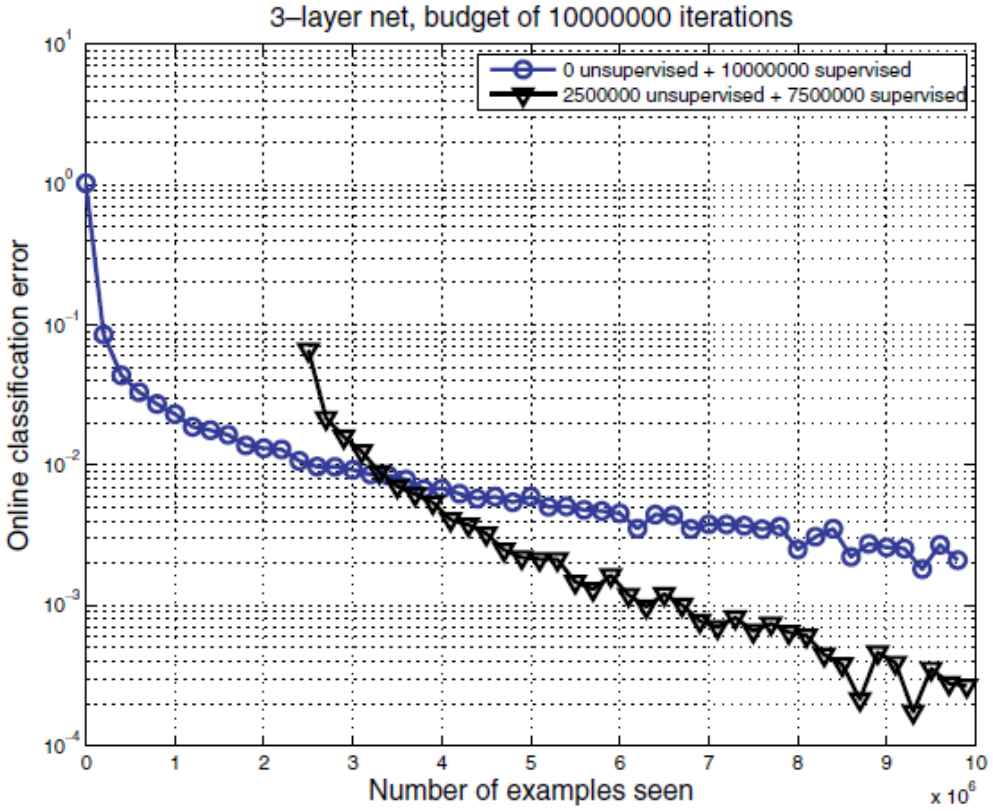

Fig 1 Deep architecture trained online with 10 million digital images, comparision shown with pre-training achieved in final better convergence- reference Yoshua Bengio (Learning Deep Architecture for AI)

4.1 BOLTZMANN MACHINE (BM)

Boltzmann machine, a stochastic recurrent neural network invented by G. E. Hinton and Terry Sejnowski, is named after the Boltzmann distribution in statistical mechanics, which is used in their sampling function. The units in a BM model are divided into 'visible' units, V, and 'hidden' units, H. The visible units are those which receive information from the 'environment', in other words, the training set is a set of binary vectors over the set V.

BM models can be seen as the stochastic, generative counterpart of Hopfield nets. The BM model is a parallel computational organization that is well suited to constraint satisfaction tasks involving large numbers of "weak" constraints. Constraint-satisfaction searches normally use "strong" constraints that must be satisfied by any solution.

This model is composed of primitive computing elements called units that are connected to each other by bidirectional links. A unit is always in one of two states, on or off, and it adopts these states as a probabilistic function of the states of its neighboring units and the weights on its links to them. The weights can take on real values of either sign. A unit being on or off is taken to mean that the system currently accepts or rejects some elemental hypothesis about the object class. The weight on a link represents a weak pairwise constraint between two hypotheses. A positive weight indicates that the two hypotheses tend to support one another; if one is currently accepted, accepting the other should be more likely. Conversely, a negative weight suggests, other things being equal, that the two hypotheses should not both be accepted. Link weights are symmetric, having the same strength in both directions.

The BM model is capable of learning the underlying constraints that characterize a class of objects by being shown samples from that object class. The network modifies the strengths of its connections so as to construct an internal generative model that produces examples with the same probability distribution as the samples it is shown. Then, when shown any particular input, the network can "interpret" it by finding values of the variables in the internal model that would generate this input and using them to generate the remainder. The learning procedure of this network will build internal representations which allow the connection strengths to capture the underlying constraints that are implicit in a large ensemble of samples taken from a class of objects.

In this network, each stochastic binary unit has an "energy" value. In this model, the global energy E is defined as follows,

$$\mathrm{E} = -\left(\sum_{i<j} w_{ij} s_i s_j + \sum_i \theta_i s_i\right),$$

where w_{ij} is the connection strength between the units *j* and *i*, and s_i is the state $s_i \in \{0,1\}$ of the unit *i* and θ_i is its bias. The connection weights have two restrictions: $w_{ii} = 0 \; \forall i$(no unit has a connection with itself), and $w_{ij}=w_{ji} \; \forall i, j$(all connections are symmetric). The weights usually are represented in matrix form $\boldsymbol{W}$ with zeros in the diagonal positions.

The difference in the global energy, denoted by ΔE_i, results from a single unit i being 0 (off) versus 1 (on) is given by

$$\Delta E_i = \sum_j w_{ij} s_j + \theta_i,$$

This can be expressed as the difference of energies of two states: $\Delta E_i = E_{i=off} - E_{i=on}$. Substitute the energy of each state with its relative probability according to the property of a Boltzmann distribution that the energy of a state is proportional to the negative log probability of that state)

$$\Delta E_i = -k_B T ln(p_{i=off}) - (k_B T ln(p_{i=on})),$$

where k_B (i.e. the Boltzmann constant) is absorbed into the artificial notion of temperature T. Rearrange terms and consider that the probabilities of the unit being on and off must sum to one:

$$\frac{\Delta E_i}{T} = ln(p_{i=on}) - ln(p_{i=off})$$
$$\frac{\Delta E_i}{T} = ln(p_{i=on}) - ln(1 - p_{i=on})$$
$$\frac{\Delta E_i}{T} = ln\left(\frac{p_{i=on}}{1 - p_{i=on}}\right)$$
$$-\frac{\Delta E_i}{T} = ln\left(\frac{1 - p_{i=on}}{p_{i=on}}\right)$$
$$-\frac{\Delta E_i}{T} = ln\left(\frac{1}{p_{i=on}} - 1\right)$$
$$exp\left(-\frac{\Delta E_i}{T}\right) = \frac{1}{p_{i=on}} - 1$$

Finally, solve for $p_{i=on}$, the probability of the unit *i* being in the "on" state.

$$p_{i=on} = \frac{1}{1 + exp\left(-\frac{\Delta E_i}{T}\right)},$$

where the scalar T is referred to as the temperature of the system. This relation is the source of the logistic function found in probability expressions in variants of the Boltzmann machine.

As for the issue of training the BM model, the distribution over the training set is denoted $P^+(V)$. The distribution over global states converges as this model reaches thermal equilibrium. We denote this distribution, after marginalize it over the hidden units, as $P^-(V)$. The training aims at approximating the "real" distribution $P^+(V)$ using the $P^-(V)$ which will be produced (eventually) by the machine. The measure of the similarity of these two distributions often uses the Kullback–Leibler divergence:

$$G = \sum_v P^+(v) ln\left(\frac{P^+(v)}{P^-(v)}\right),$$

where the sum is over all the possible states of ***V***, ***G*** is a function of the weights, since they determine the energy of a state, and the energy determines $P^-(v)$, as promised by the Boltzmann distribution. A gradient descent algorithm can be used over ***G***, so a given weight, w_{ij} is changed by subtracting the partial derivative of ***G*** with respect to the weight.

Boltzmann machine training consists of two alternative phases and switches iteratively between them. In the "positive" phase, the states of visible units are clamped to a particular binary state vector sampled from the training set, according to $\boldsymbol{P}^+$. In the "negative" phase, the network is allowed to run freely—no units have their states determined by external data. The gradient with respect to a given weight w_{ij} is given by the equation

$$\frac{\partial G}{\partial w_{ij}} = -\frac{1}{R}[p_{ij}^+ - p_{ij}^-],$$

where $\boldsymbol{P}_{ij}^+$ is the probability of units i and j both being on, when the machine is at equilibrium in the positive phase, $\boldsymbol{P}_{ij}^-$ the probability of units i and j both being on when the machine is at equilibrium in the negative phase, and ***R*** the learning rate. This result follows from the fact that at thermal equilibrium the probability $\boldsymbol{P}^-(\mathbf{s})$ of any global state ***s*** when the network is free-running is given by the Boltzmann distribution. This learning rule is fairly biologically plausible because the only information needed to change the weights is provided by "local" information. The synapse connection weight needs no information about anything other than the two neurons it connects.

The training of BM models does not use the EM algorithm, which is commonly used in machine learning. Minimizing the KL-divergence is equivalent to maximizing the log-likelihood of the data. The training procedure therefore performs gradient ascent on the log-likelihood of the observed data. In contrast, the EM algorithm must calculate the posterior distribution of the hidden nodes before the maximization of the expected value of the complete data likelihood during the M-step. Training the biases is similar, but uses only single node activity:

$$\frac{\partial G}{\partial \theta_{ij}} = -\frac{1}{R}[p_i^+ - p_i^-].$$

BM is one of the examples of an ANN model capable of learning internal representations, and is able to represent and solve difficult combinatoric problems, given sufficient training time. The BM model is capable of learning the underlying constraints that characterize a class of objects by being shown samples from that object class. This network modifies the strengths of its connections so as to construct an internal generative model that produces examples with the same probability distribution as the samples it is shown. Then, when shown any particular input, the network can "interpret" it by finding values of the variables in the internal model that would generate this input and using them to generate the

remainder. The learning procedure of this network will build internal representations which allow the connection strengths to capture the underlying constraints that are implicit in a large ensemble of samples taken from a class of objects. For example, if trained on the human face images, this model will theoretically model the distribution of these images, and could use that model to complete a partially occluded image. However, due to a number of issues, BM models with unconstrained connectivity have not proven useful for practical problems in machine learning. These models are theoretically intriguing, due to the locality and Hebbian nature of their training algorithm, the parallelism, and the resemblance of their dynamics to simple physical processes. In addition, the time the machine must be run in order to collect equilibrium statistics grows exponentially with the machine's size, and with the magnitude of the connection strengths. Furthermore, connection strengths are more plastic when the units being connected have activation probabilities intermediate between zero and one, leading to a so-called variance trap. The net effect is that noise causes the connection strengths to follow a random walk until the activities saturate.

It was conjectured that the learning can be made efficient enough to be useful for practical problems, it the connectivity across the units is constrained. One of attempts developed Restricted Boltzmann Machine (RBM) in which intra-layer units have no connection.

4.2 RESTRICTED BOLTZMANN MACHINE (RBM)

4.2.1 THE NETWORK ARCHITECTURE

Hinton et al. proposed an algorithm for learning deep networks (e.g. deep belief network and MLP) by treating each layer as an RBM and greedily training the network one layer at a time from the bottom up. An RBM is a two layer recurrent neural network in which stochastic binary inputs are connected to stochastic binary outputs using symmetrically weighted connections. The first layer corresponds to inputs (a set of visible units **v**) and another layer to a set of hidden units **h**, shown as the following figure. The weight matrix **W** represents the symmetric interaction terms between a visible unit and a hidden unit. Finally, the $\boldsymbol{b}$ and $\boldsymbol{a}$ represent the bias terms of the hidden layer and the visible layer, respectively.

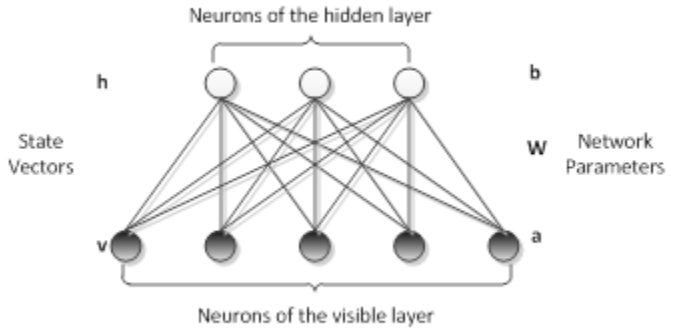

Fig 1. the diagram of the RBM

RBM is an energy-based undirected generative model that uses the hidden layer variables to model a distribution over visible variables. It is restricted in the sense that there are no visible-visible or hidden-hidden connections. The undirected model for the interactions between the hidden and visible variables is used to ensure that the contribution of the likelihood term to the posterior over the hidden variables is approximately factorial which greatly facilitates

inference. Energy-based model means that the probability distribution over the variables of interest is defined through an energy function.

4.2.2 THE NETWORK TRAINING ALGORITHM

RBM can be trained by using the standard BM learning algorithm which follows a noisy but unbiased estimate of the gradient of the log likelihood of the data. One way to implement this algorithm is to start the network with a data vector on the visible units and then alternate between updating all hidden units in parallel and updating all visible units in parallel. Each update picks a binary state for a unit from its posterior distribution given the current states of all the units in the other set. The Gibbs sampling runs alternatively between the two sides, until the network runs to an equilibrium state. This searching can be simplified by updating the weights to minimize the Kullback-Leibler divergence, $Q^0||Q^\infty$, between the data distribution Q^0 and the equilibrium distribution of fantasies over the visible units Q^∞ produced with $\Delta w_{ij} = \varepsilon (<s_i s_j>_{Q0} - <s_i s_j>_{Q\infty})$, where $<s_i s_j>_{Q0}$ is the expected value of $s_i s_j$ when data is clamped on the visible units and the hidden states are sampled from their conditional distribution given the data, and $<s_i s_j>_{Q\infty}$ is the expected value of $s_i s_j$ after prolonged Gibbs sampling. However, this learning rule does not work well because it takes a long time to approach thermal equilibrium and the sampling noise in the estimate of $<sisj>_{Q\infty}$ can swamp the gradient. It was shown that it is far more effective to minimize the difference between $Q^0||Q^\infty$ and $Q^1||Q^\infty$, where Q^1 is the distribution of the one-step reconstructions of the data that are produced by first picking binary hidden states from their conditional distribution given the data and then picking binary visible states from their conditional distribution given the hidden states. The exact gradient of this "contrastive divergence" is complicated because the distribution Q^1 depends on the weights, but this dependence can safely be ignored to yield a simple and effective learning rule for following the approximate gradient of the contrastive divergence: $\Delta w_{ij} = \varepsilon(<s_i s_j>_{Q0} - <s_i s_j>_{Q1})$. More detailed description can be found in the work made by Hinton et al.

This model is characterized by the energy function defined as:

$$E(v,h) = -\mathbf{h}^T\mathbf{W}\mathbf{v} - \mathbf{b}^T\mathbf{v} - \mathbf{c}^T\mathbf{h} \quad (1)$$

where **W** is the weight matrix and **b**, **c** are the bias vectors for visible and hidden layers, respectively. The layer to layer conditional distributions are:

$$P(v_i=1|\mathbf{h}) = \sigma(b_i + \Sigma_j W_{ji} h_j) \quad (2)$$

$$P(h_j=1|\mathbf{v}) = \sigma(c_j + \Sigma_j W_{ji} v_i) \quad (3)$$

where $\sigma(x) = (1+e^{-x})^{-1}$ is the logistic function whose output range is in the (0,1) range. By sampling with the above probabilities, the output (0 or 1) of an RBM unit is determined. The following is the description of how to train an RBM and how to use it in the construction of a DBN or other deep networks.

- First, it should be noticed that RBM training is unsupervised. Given a training example, ignore its class label and propagate it stochastically through the RBM. The outputs of the hidden units follow the conditional distribution specified in equation (3).
- Second, sample from this distribution to produce a binary vector. This vector is propagated in the opposite direction through the RBM (from hidden units to visible units using equation (2)) which results in a "confabulation" (reconstruction) of the original input data.
- Update the state of the hidden units by propagating this confabulation through the RBM.
- Repeat the above steps for all training examples, and update **W**, **b**, and **c**.
- Repeat the above procedure for a defined number of epochs till the reconstruction error of the original data becomes small, i.e. the confabulation is very similar to the input data.

It can be seen that RBM is an undirected graphical model which use hidden/latent variables in an energy-based model to achieve highly expressive marginal distributions. Although maximum likelihood learning is intractable in this model, the Contrastive Divergence (CD) algorithm has been shown to be effective in training RBM to model a variety of high dimensional data distributions, e.g. images and image transformations. Generally, the CD-1 algorithm gives satisfactory training results.

Originally, RBM was proposed to process stochastic binary inputs, which has restricted applications. A variant is further suggested, i.e. Gaussian-Bernoulli Restricted Boltzmann Machine (GB-RBM). A GB-RBM models the probability density of continuous-valued data using binary latent variables. It consists of a layer of Gaussian visible units that correspond to components of data vectors and a hidden layer of binary units. Each unit is still connected to all units in the other layer. The latent variables of a learnt GB-RBM can be used as meaningful unsupervised features.

4.2.3. THE PHYSICAL MEANING

Mathematically, RBM is a special type of Markov random field that has one layer of (typically Bernoulli) stochastic hidden units and one layer of (typically Bernoulli or Gaussian) stochastic visible or observable units. RBMs can be represented as bipartite graphs, where all visible units are connected to all hidden units, and there are no visible-visible or hidden-hidden connections. Careful training RBMs is essential to the success of applying RBM and related deep learning techniques to solve practical problems.

The physical significance of RBM models is stated as follows. RBM, a probabilistic model, models a distribution by splitting the input space in many different ways. It is similar to Principal Components Analysis (PCA), in that it is trained by unsupervised learning to capture leading variations in the data, and it yields a new representation of the data. Each hidden unit is associated with a binary random variable indicating which side of a hyper-plane the input vector locates in. The probability for that binary variable to turn on or off depends on how far the input vector is from that hyper-plane. Each hidden unit performs a clustering involving just two classes, or it can be considered as a binary attribute automatically discovered during the learning, to explain the dependencies between the elements of the input vector. The configuration of hidden units indicates a region in the input space, and the vector of probabilities associated with each hidden unit is a generally faithful representation of the location of the input vector in the input space. There are intimate relations between RBMs and auto-encoders. Hence, that vector of probabilities can be used as a novel, more abstract, representation of the raw input vector. It tends to separate some of the factors that explain the variations present in the training set.

Unlike other unsupervised learning algorithms (e.g. clustering), RBMs discover a “rich’ representation of the input data. Whereas you would need a huge number of clusters to capture all the variations in the input (with ordinary clustering algorithms), you can get away with a reasonable small RBM and capture complicated distributions, because N hidden units can represent up to 2^N different regions in the input space. With ordinary clustering you would need O(2^N) parameters (and examples) to capture many regions, whereas with RBMs you only need O(N) parameters. It works because (or if) there is some structure (to be captured) in the input distribution.

4.2.4 THE FOLLOWCHART AND PSEUDO CODE

As mentioned in the foregoing sub-section, the CD-1 algorithm has been proposed to train an RBM (i.e., set the hyper-planes associated with each hidden unit = discover these attributes or directions of variation), and this model approximates or relates to gradient descent on the log-likelihood of the data, trying to model the distribution of the input dataset. The follow chart is presented in the figure with the pseudo-code thereafter. Details of how to train an RBM model can be found in the manual written G. E. Hinton et al.

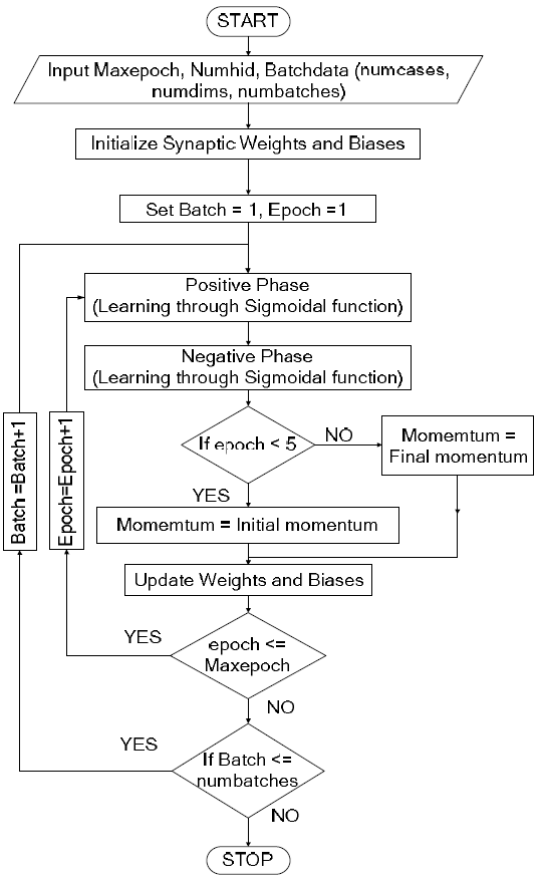

Fig 2. Iterative Learning Process of the RBM model.

The pseudo-code of the RBM learning algorithm is given as follows.

-Visible neurons initially set to a batch of training examples, denoted vis_batch_0

-Repeat until convergence{

I) Sample hid_batch_0 from P(h|vis_batch_0)

*a) tmp_matrix_1 = vis_batch_0 * weights*

b) tmp_matrix_2 = tmp_matrtx_1 + hid_biases

c) tmp_matrix_3 = sigmoid(tmp_matrix_2)

d) hid_batch_0 = tmp_matrix_3 > rand()

2) Sample vis_batch_1 from P(v | hid_batch_0)

3) Sample hid_batch_1 from P(h | vis_batch_1)

4) Update parameters:

*a) weights += α(vis_batch_0^T * hid_batch_0 - visbatch_ 1^T * hid_batch_1)*

*b) visbiases += α (vis_batch_0^T * **1** - visbatch_ 1^T* **1**)*

*c) hidbiases += α (hid_batch_0^T * **1** - hid_batch_1^T * **1**)*

}

The pseudo-code shows that each sampling process is essentially matrix-matrix multiply between a batch of training examples and the weight matrix, followed by a neuron activation function, which in many cases is a sigmoid function ($1/(1 + e^{-x})$). The sampling between the hidden and visible layers is followed by a slight modification in the parameters (controlled by the learning rate α) and repeated for each data batch in the training set, and for as many epochs as is necessary to reach convergence.

The RBM training is dominated by matrix-matrix multiplication of batches of neuron values by a weight matrix **W**. In addition, a significant amount of computation time is spent in the matrix-matrix multiplication of the visible neuron batches V and hidden neuron batches H in the weight update phase. To exploit the parallelism in matrix multiply operations, a hardware RBM implementation should maximize the number of multipliers that can be supported by the memory bandwidth and logic gate resources, while reserving resources for other computation such as adders or the sigmoid function. The training algorithm requires three matrix-matrix products: VW, HW^T, and V^TH. Given M multipliers, to fully utilize the M multiplier resources, M weights must be read from the memory blocks each cycle.

A scalable approach involves changing the ordering of the matrix multiplication operations for different computation phases so that only row vectors of the weight matrix need to be accessed. The cost is the use of twice as many adders. The matrix multiplication HW^T can be performed as a collection of vector inner products, where the row vectors of the weight matrix W are accessed in each cycle. To obtain the product VW, W^TV^T is computed instead, which is viewed as a collection of vectors, each of which is a linear combination of the column vectors of W^T, with V^T as the coefficients. Thus, only columns of W^T (rows of W) need to be read from the memory, which are all multiplied by an element of a row vector from V each cycle. Another benefit is that in the weight update phase, the matrix product V^TH can reuse the structure for multiplication VW. The memory bandwidth requirement for visible neurons is one neuron per cycle to be broadcasted, while the bandwidth requirement for a hidden neuron computation is a row vector of the hidden neuron batch H. If the V^TH is viewed as the sum of the outer products of each row vector of V and H, the visible node can be broadcasted in each cycle, which is multiplied by a row of hidden neurons to get the outer product. Then, the outer products can be summed by using the existing accumulator. As the matrix multiplication structure for weight updates is basically the same as the hidden neuron computation phase, the main focus is put upon the two multiplications VW and HW^T.

4.2.5 THE USE IN THE DEEP ARCHITECTURE

RBM is often used as a building block for deep models. The benefit of using the RBM models is that they often provide a good initialization for feed-forward neural networks, and they can effectively utilize large amounts of unlabeled data, which has led to the success in a variety of application domains. Once a layer of the network is trained, the parameters w_{ij}, b_j, c_i's are frozen and the hidden unit values given the data are inferred. These inferred values serve as the "data" used to train the next higher layer in the network. Hinton et al. showed that by repeatedly applying such a procedure, one can learn a multilayered deep belief network. In some cases, this iterative "greedy" algorithm can further be shown to be optimizing a variational bound on the data likelihood, if each layer has at least as many units as the layer below (although in practice this is not necessary to arrive at a desirable solution).

A main source of tractability in the RBM model is that, given an input, the posterior distribution over hidden variables is factorizable and can be computed and sampled from. This model is a density model for feature extraction. After training, hidden units can be considered to act as feature detectors—they form a compact representation of the input vector. RBM models however tend to learn distributed, non-sparse representations. Sparsity and competition in the hidden representation is beneficial, and while an RBM model with competition among its hidden units would acquire some of the attractive properties of sparse coding, such constraints are typically not added, as the resulting posterior over the hidden units seemingly becomes intractable. In addition, despite the benefits of the RBM model, there is a disconnection between its unsupervised nature and the final discriminative task (e.g., classification) for which the learned features are used. This disconnection has motivated the search for ways to improve task-specific performance, while still retaining the unsupervised nature of the original model.

Several RBM models can be stacked on top of each other such that higher level RBMs learn to model the posterior distributions of the hidden variables of the lower level RBM models. This stacking process has the property that, under certain conditions, adding another RBM to the stack creates a new composite model, called a Deep Belief Net (DBN) that has a better lower bound on the log probability of the training data than the previous DBN. The stacking procedure is as follows. After learning a Gaussian-Bernoulli RBM (for applications with continuous features such as speech) or Bernoulli-Bernoulli RBM (for applications with nominal or binary features such as black-white image or coded text), the activation probabilities of its hidden units are treated as the data for training the Bernoulli-Bernoulli RBM one layer up. The activation probabilities of the second-layer Bernoulli-Bernoulli RBM are then used as the visible data input for the third-layer Bernoulli-Bernoulli RBM, and so on. Similarly, other kinds of deep models can be built up.

4.2.6. POTENTIAL IMPROVEMENTS FOR PERFORMANCE

A method for improving performance is to incorporate sparsity into the learned representation. In the context of computer vision, sparsity has been empirically linked with learning features invariant to local transformations, and sparse features are often more interpretable than dense representations after unsupervised learning. For directed models (e.g. sparse coding), sparsity can be enforced using a Laplace or spike-and-slab prior. For undirected models, the direct introduction of hard sparsity constraints into the energy function often results in non-trivial dependencies between hidden units. This makes inference intractable, and a way around this issue is to encourage sparsity during training by using a penalty function on the expected conditional hidden unit activations given data. The training-time procedure however is a heuristic and does not guarantee sparsity at test time.

4.3 AUTO-ENCODER

An auto-encoder is a feed-forward neural network which is trained to approximate the identity function. This model is designed with the intention to reconstruct the input data at the output layer, and the reconstructed results are expected to be close to the input. The main difference between auto-encoders and traditional neural networks is the size of the output layer. In an auto-encoder, the output layer has the same dimensionality as the input, visible layer, while the hidden layers (especially the deepest hidden layer) are generally smaller than the input and output layers in size, so the dimensionality of the input data is reduced to a low-dimension code space at the hidden layers. Like PCA, auto-encoders give mappings in directions between the data and the code space.

Sigmoid activation functions are normally used in auto-encoders for non-linear mapping. If linear activation functions are used, auto-encoders will perform similar to PCA. Although the auto-encoder model also uses the feed-forward architecture, its training does not require the labeled training data, which is quite different from other neural networks of using the same architecture. During the training procedure, the network is trained to map from a vector of values to the same vector. When used for dimensionality reduction purposes, one of the hidden layers is limited to contain only a small number of units. The network must learn to encode the vector into a small number of dimensions and then decode it back into the original space. Accordingly, the first half of the network is a model which maps from high to low-dimensional space, and the second half maps from low to high-dimensional space. The following figure shows this architecture vividly.

Previously, auto-encoders were trained under supervision by the back-propagation algorithm, and the networks can be trained by minimizing the mean square error between the original and the reconstructed data. The required gradient is easily obtained by using the chain rule to back propagate the error derivatives first through the decoder network and then through the encoder network. However, the training of deep auto-encoders has only recently become possible through the use of RBM models and stacked denoising auto-encoders.

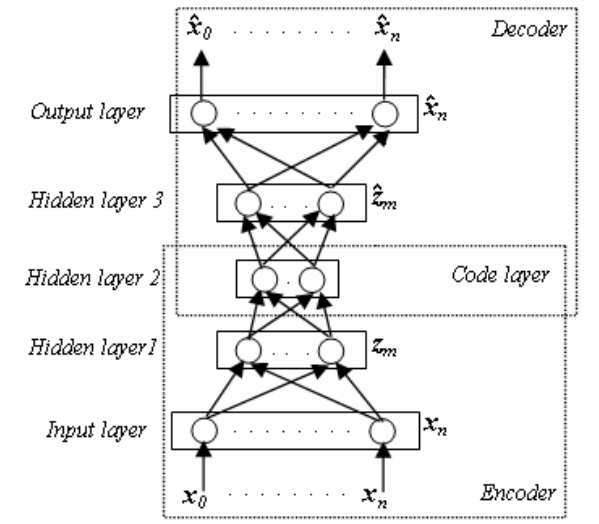

Fig 3. The architecture of a multilayer auto-encoder neural network

5 TRAINING RBM AND AUTO-ENCODER

Neural networks are able to extract complex underlying structure from the statistical distribution of the input data, because of the architecture consisting of parallel, simple computing elements. Deep networks enjoy new success in many applications with a large number of neurons in multiple hidden layers, which often results in millions or even billions of free parameters. With the unprecedented growth of commercial and academic data sets in recent years, there occurs a surge in interest in effective, scalable parallel algorithms for training deep models, including DBN, stacked de-noising auto-encoder, CNN, and the classifiers based on sophisticated feature extraction techniques. Correspondingly, large-scale deep learning involves both large volumes of data and large models in nature.

5.1 ISSUES COMMON TO THE DEEP NETWORKS TRAINING

Compared to the networks of shallow architecture where few parameters are preferable to avoid over-fitting problems, it is not a trivial task at all to train large-scaled deep networks, due to the fact that iterative computations inherent in most learning algorithms are extremely difficult to be parallelized. As a result, software programs of large-sized networks (e.g. RBM models with 2000×500 nodes) are unable to satisfy the real-time constraints required to solve real-world problems. Large-scale deep learning therefore has to be accelerated by employing powerful hardware techniques in computation. Existing implementations for deep learning are designed chiefly to utilize the limited computation resources offered by modern multiple X86 CPU cores to the full. For example, one way to scale up DBN models is to use multiple CPU cores, with each core dealing with a subset of training data (data-parallel schemes). Other performance-improving techniques have been used, including carefully designing data layout, batching of the computation, and leveraging SSE2, SSE3, and SSE4 instructions. Some accelerated implementations explore FPGA-based hardware architecture, which used a control unit implemented in a CPU, a grid of multiple full-custom processing tiles, and a fast memory. In fact, each processing element usually utilizes only a small fraction of the processor's resources, exacerbating the performance bottleneck and limiting its cost effectiveness. Recently, some newly-developed frameworks make advantage of the mammoth computing power available in the GPU devices, because of their particular suitability for massively parallel computing with more transistors devoted for data processing work. These implementations are exclusively based on the NVIDIA products and CUDA. These efforts mainly focus on optimizing the GPU computation, but

do not pay much attention to data transmission between the CPU and the GPU devices and other problems. The efficient solution for these problems highly possibly requires the computing system with the heterogeneous architecture.

Besides the complexity of deep models, the training usually needs ten or even fifty hyper-parameters, depending on how these models are parameterized and how many hyper-parameters are chosen to fix at a reasonable default. Hyper-parameter optimization is another problem of optimizing a loss function over a graph-structured configuration space. The difficulty of tuning deep models makes published results difficult to reproduce and extend, and makes the original investigation of these models seem more like an art than a science. In some research results, it was demonstrated that the challenge of hyper-parameter optimization in large-scaled deep models is a direct impediment to scientific progress. New results have advanced state-of-the-art performance on image classification problems by more concerted hyper-parameter optimization in simple algorithms, rather than by innovative modeling or machine learning strategies.

It would be wrong to conclude that feature learning is useless. Instead, hyper-parameter optimization should be regarded as a formal outer loop in the learning process. A learning algorithm, as a functional from data to classifier (taking classification problems as an example), includes a budgeting choice of how many CPU cycles are to be spent on hyper-parameter exploration, and how many CPU cycles are to be spent evaluating each hyper-parameter choice (i.e. by tuning the regular parameters). It has been suggested that with current generation hardware (e.g. large computer clusters and GPUs) the optimal allocation of CPU cycles includes more hyper-parameter exploration than has been typical in the machine learning literature. Deep networks, contrary to the common belief, can be trained successfully by back-propagation. Despite the numerous free parameters, deep networks seem to learn faster (better recognition rates after fewer epochs) than shallow ones. The above comments have been mentioned in some research reports.

5.2 PRE-TRAINING

It is a challenging problem to train deep networks. These networks, since they are built from the composition of several layers of non-linearity, yield an error surface that is non-convex and hard to optimize, with the suspected presence of many local minima. A gradient-based optimization procedure would thus end in the local minimum of whatever basin of attraction the minimum searching started from.

The newly-developed learning algorithms for deep models are based on a similar approach: greedy layer-wise unsupervised pre-training followed by supervised fine-tuning. The central idea is to learn a hierarchy of features one level at a time, using unsupervised feature learning to learn a new transformation at each level to be composed with the previously learned transformations; essentially, each iteration of unsupervised feature learning adds one layer of weights to a deep network. Finally, the set of layers could be combined to initialize a deep supervised predictor (e.g. a neural network classifier) or a deep generative model (e.g. a DBN). In the pre-training stage, each layer learns a nonlinear transformation of its input (i.e., the output of the previous layer) that captures the main variations in its input. This unsupervised pre-training only sets the stage for a final training phase where the deep architecture is fine-tuned with respect to a supervised training criterion with a gradient-based optimization.

Evidence has been found that pre-training is especially helpful in optimizing the parameters of the lower-level layers. The mean test error and its variance are reduced with pre-training for sufficiently large models. This effect is more pronounced for deeper models. It has been proposed that pre-training is a kind of regularization mechanism, by minimizing variance and introducing a bias towards configurations of the parameter space that are useful for unsupervised learning. Interestingly, pre-training seems to hurt performance for shallow networks with small-sized layers. It has also been verified that unsupervised pre-training does something rather different than induce a good initial marginal distribution. From this perspective, the advantage of pre-training could be that it puts us in a region of

parameter space where basins of attraction run deeper than when picking starting parameters at random. A brief summary of main advantages is given as follows. A detailed description and the tests can be found in the literature.

1 **Better generalization**. Unsupervised pre-training gives substantially lower test classification error than no pre-training, for the same depth or for smaller depth on various vision datasets, when choosing the number of units per layer, the learning rate, and the number of training iterations to optimize the classification error on the validation set.

2 **Aid to optimization**. In some experiments, the training error of the trained classifiers is low, with or without pre-training. This finding would make it difficult to distinguish between the optimization and regularization effects of pre-training. Some researchers hypothesized that higher layers in the network were over-fitting the training error. To make it clearer whether some optimization effect (of the lower layers) was going on, they constrained the top layer to be small (20 units instead of 500 or 1000). The comparison tests showed that final training errors are higher without pre-training.

3 **Distinct local minima**. With different random initializations, with or without pre-training, each trajectory ends up in an apparently different local minimum corresponding to different parameters and a different function. It is difficult to guarantee that these are indeed local minima, but the test results are consistent with that interpretation by using visual inspection of trajectories in function space, estimation of second derivatives in the directions of all the estimated eigenvectors of the Jacobian. The regions in function space reached without pre-training and with pre-training seem completely disjoint (i.e. no model without pre-training ever gets close to a model with pre-training). Nevertheless, in some studies, the performance of their networks with pre-training is similar to the standard back-propagation without pre-training. This observation can be explained by the fact that the networks without pre-training already performs well on their tasks and may have reached the limit of the given architecture. It might be the case that pre-training puts us in a region of parameter space in which training error is not necessarily better than when starting at random (or possibly worse), but which systematically yields better generalization (test error).

4 **Lower variance**. In the same set of experiments, the variance of final test error with respect to the initialization random seed is larger without pre-training, and this effect is magnified for deeper architectures. This supports a regularization explanation, but does not exclude an optimization hypothesis either. It was claimed that pre-trained networks lead to less deviation in some studies.

5 **Capacity control**. When all the layers are constrained to a smaller size, the advantage gained by pre-training will disappear, and the generalization becomes worse with pre-training for very small-sized layers. This finding is compatible with a regularization effect and seems incompatible with a pure optimization effect.

6 **Support for small-sized training sets.** Pre-training is helpful for the cases where small-sized training sets are available. At a same training cost level, pre-trained models systematically yield a lower test cost than randomly initialized ones. The advantage appears to be one of better generalization, rather than merely a better optimization procedure. In this sense, pre-training appears to have a similar effect to that of a good regularizer or a good “prior” on the parameters, even though no explicit regularization term is apparent in the cost being optimized. It might be reasoned that restricting possible starting points in parameter space to those that minimize the pre-training criterion does in effect restrict the set of possible final configurations for parameter values.

The greedy layer-wise unsupervised strategy provides an initialization procedure, and thereafter deep networks are fine-tuned to the global supervised objective. Simply put, three principles for training deep networks are: (1) pre-train one layer at a time in a greedy way; (2) use unsupervised learning at each layer in a way that preserves information from the input and disentangles factors of variation; and (3) fine-tune the whole network with respect to the ultimate criterion of interest. The idea of using unsupervised learning at each hidden layer is that the network first learns simple concepts, on which it builds more abstract concepts. Upper layers are supposed to represent more “abstract” concepts which explain the input observation, whereas lower layers extract “low-level features” from the input. However, an issue of the RBM

training is the lack of a clear criterion to stop the pre-training. Typically, the mean squared error between the input and its reconstruction result, an indicator how well the reconstruction is done, is used to measure the training performance.

5.3 ISSUES SPECIFIC TO THE AUTO-ENCODER TRAINIGN

An auto-encoder is often trained using the back-propagation (BP) algorithm and its variants. The BP algorithm, the most widely used algorithm involved in neural network training, is a supervised training algorithm based on the error-correction learning rule, which consists of two consecutive parameter tuning passes (forward and backward, respectively) through the layers of the neural network. This algorithm has a good property that it allows neural networks to be expressed and debugged in a modular fashion. For instance, it can be assumed that a module M has a forward propagation function which computes its output M(I,W) as a function of its input I and its parameters W, and has a backward propagation function (with respect to the input) which computes the input gradient as a function of the output gradient, a gradient function (with respect to the weight), which computes the weight gradient with respect to the output gradient, and a weight update function, which adds the weight gradients to the weights using some updating rules, including batch, stochastic, momentum, weight decay, etc.

The BP algorithm however does not guarantee that the procedure of tuning neural network parameters converges towards a global minimum, instead of local ones. In addition, another issue with this algorithm happens in training the networks with many hidden layers. The errors back-propagated to the first layers are minuscule and quite ineffectual, which causes the network to almost always learn to reconstruct the average of all the training data. Although advanced BP variant algorithms (e.g. the conjugate gradient method) help with this in some degree, it still results in very slow learning and poor solutions. This problem is remedied by using initial weights that approximate the final solution. The process to find these initial weights is often called pre-training. In this project, the pre-training involves the RBM model, which was proposed by G.E. Hinton and et al and briefly described in the forgoing section.

5 The GPU-ACCLERATED IMPLEMENTATION OF RBM and AUTO-ENCODER

Deep networks are known to be difficult to train, because of the fact that many hyper-parameters affect the behavior of the model. The task of separating a given class from all other classes might be best achieved by learning the optimal feature for the samples belonging to that particular class, rather than by training on the average separation performance for all classes. It is normally time consuming and difficult to design a new feature-learning model for every object class, but deep learning requires very little modification

6.1 ADVANTAGES FOR USING THE GPU

Fundamentally, the CPU and the GPU are built based on very different philosophies. CPUs are designed to provide fast response times to a task and for a wide variety of applications. In addition to frequency scaling, architectural advances, e.g., branch prediction, out-of-order execution, and super-scalar, have been responsible for performance improvement. These advances however come at the price of increasing complexity/area and power consumption. As a result, main stream CPUs today can pack only a small number of processing cores on the same die to stay within the power and thermal envelopes. By contrast, GPUs are built specifically for rendering and other graphics applications that have a large degree of data parallelism (each pixel on the screen can be processed independently). Graphics applications are latency tolerant (the processing of each pixel can be delayed as long as frames are processed at interactive rates). As a result, GPUs can trade off single thread performance for increased parallel processing. For instance, GPUs can switch from processing one pixel to another when long latency events (memory accesses) are encountered and can switch back to the former pixel at a later time. This approach works well when there is ample data-level parallelism. The speedup of

an application on GPUs is ultimately limited by the percentage of the scalar section (in accordance with Amdahl's law). Finally, current GPUs are suitable for streaming computations.

As for deep learning, the GPU can bring the following benefits.

1 Dense matrix operations are frequently used in the process of updating the network weights and bias vectors of hidden layers. These operations are dense fine-grained and abundant parallel computation, and exhibit regular memory access patterns, so these operations are natural candidates for the GPU. As for the matrix-matrix multiplication, three optimization strategies can be used to exploit the potential of GPU computing capabilities, i.e. wider memory operations, double-buffering in local memory, and instruction scheduling. AMD has provided OpenCL implementations of dense matrix operations, including SGEM.

2 The computational unit of a deep network is a linear projection followed by a point-wise non-linearity, typically a logistic function. These functions can be implemented independently in parallel by using processing elements in the GPU devices.

3 The GPU implementation scales well with the sizes of DNN models. For the cases where small-sized networks are concerned, the speedup time is small since these networks fit better inside the CPU cache, and GPU resources are underutilized. For those large-sized networks, the GPU-based implementation runs much faster than a compiler-optimized CPU version. Given the flexibility of the current GPU characteristics, there usually is a significant speedup.

4 There are no well-established rules to determine the architecture. The number of hidden layers and the number of nodes in the hidden layers determine the basic neural network architecture. Selecting good network architecture to achieve the best detection and classification results is therefore an open problem. A commonly used approach is to try different combination of parameters in an ad hoc manner and empirically select the "best" architecture based on test results. However, the manual optimization process usually only searches very limited regions of the large-dimensional parameter space and it is not systematic. This demands computing power, which is supported well by the systems integrated with GPU devices.

5.2 ADVANTAGES OF USING OPENCL

Although the computation of deep learning consists mainly of densely linear algebra operations, including the matrix-matrix multiplication and matrix addition, which are in nature suitable for the massively parallel computing devices, the computation also requires various kinds of serial operations, which the CPU does well in. Fast and energy-efficient implementations of deep learning are heterogeneous, and accordingly require the hybrid computing platforms. In addition, future designs of microprocessors and large HPC systems will be hybrid/heterogeneous in nature, relying on the integration (in varying proportions) of major types of components:

1. Multi-cores CPU technology, where the number of cores will continue to escalate while avoiding the power wall, instruction level parallelism wall, and the memory wall; and

2. Special purpose hardware and accelerators, especially GPUs, which are in commodity production, have outpaced standard CPUs in performance, and have become as easy, if not easier to program than multicore CPUs.

OpenCL can support the above requirements because of its features, compared to the NVIDIA VUDA.

6.2 OPTIMIZATION STRATEGIES FOR THE OPENCL IMPLEMENTATION of Restricted BOLTZMANN MACHINE

According to the foregoing description, the RBM training algorithm is dominated by matrix multiplication. It was claimed that the percentage of runtime consumed in matrix multiplication is generally over 90%, especially for the large-sized networks. Hence, this algorithm can run considerably faster by accelerating the matrix-matrix multiplication. This

promising attribute can be implemented because of the presence of fine-grain parallelism, which can be supported very well by the GPU devices.

Because it is time consuming to transfer the data forth and back between the host and the GPU, one needs to minimize host-device transfers and take advantage of the memory hierarchy in the GPU devices. To achieve this, a strategy is to store all parameters and a large chunk of training examples in the global memory during training. This will reduce the overhead of transferring data between the two sides and allow for parameter updates to be carried out fully inside GPUs. In addition, to utilize the MP/SP levels of parallelism, a few of the unlabeled training data in global memory will be selected each time to compute the updates concurrently across blocks (data parallelism).

6.2.1 GENERATION OF PSEUDO-RANDOM NUMBERS

When implementing the RBM learning, Gibbs sampling is repeated. This can be implemented by generating two sampling matrices P(h|x) and P(x|h), with the (i,j)-th element P(hj|xi) (i.e., the probability of j-th hidden node given the i-th input example) and P(xj|hi), respectively. The sampling matrices can be implemented in parallel for the GPU, where each block takes an example and each thread works on an element of the example. Similarly, the weight update operations can be performed in parallel using linear algebra packages for the GPU after new examples are generated.

It is not appropriate for the CPU to generate pseudo random numbers and transmit them to the GPU to complete the Gibbs sampling. This approach is not efficient because of the RBM training needs a large number of sampling operations. It will also heavily influence the training time. An alternative solution uses the GPU to generate pseudo random numbers directly. In this work, the method proposed by J.K.Salmon et al was used. The details can be found in their paper.

6.2.2 RANDOM PERMUTATION OF TRAINING DATASET

The training of deep networks requires the data in the training set should be permutated in a random ordering for each training epoch. The dataset usually has a mammoth of data, e.g., hundreds of Giga bytes in size, which has to be kept in individual files. It therefore is impractical to load all the data into the host memory for random permutation.

Given the training vectors $v_1,\ldots,v_N$, the generation of a random sequence takes a random number between 1 and N sequentially, ensuring that there is no repetition, in order to generate a random ordering uniformly, i.e., each of the N! permutation results happens in the same probability. For clarity, the permutation is interpreted as a two-line notation shown as follows.

$$\begin{pmatrix} v_1 & v_2 & v_3 & \cdots & v_N \\ x_1 & x_2 & x_3 & \cdots & x_N \end{pmatrix},$$

where the permutated sequence is denoted by $(x_1,\ldots, x_N)$. This native method will require occasional retries whenever the random number picked is a repeat of a number already selected.

In this project, a simple algorithm was used to generate a permutation of N items uniformly at random without retries, called the Knuth shuffle, is to start with any permutation (for example, the identity permutation), and then go through the positions 1 through N−1, and for each position i swap the element currently there with a randomly chosen element from positions i through N, inclusively. It's easy to verify that any permutation of N elements will be produced by this algorithm with probability exactly 1/N!, which yields a uniform distribution over all permutations. The initialization to the identity permutation and the shuffling may be combined, as in the following example code. It requires a function uniform(m) which returns a random integer between 0 and m, inclusively.

unsigned uniform(unsigned int m); / return a random integral number in the range of 0 to m, inclusively */*
unsigned permute(unsigned int perm[], unsigned int n)

```
{
   unsigned i;
   for (i = 0; i < n; i++)
   {
      unsigned j = uniform(i);
      perm[i] = perm[j];
      perm[j] = i;
   }
}
```

Note that the first assignment to perm[i] might be copying an uninitialized value, if j happens to be equal to i. However, in this case, it is immediately overwritten with a defined value on the next line. In addition, the uniform() function cannot be implemented simply as random() % (m+1) unless a bias in the results is acceptable.

In the OpenCL implementation, the code does not permute the training vectors in a direct fashion. The code gives each vector a unique index number within the dataset and permutes these index numbers beforehand. Thereafter, the code uses the permutated index numbers to shuffle those vectors accordingly. For the sake of efficiency, the code separates this vector permutation into three steps: (1) the code shuffles the vectors in each source file individually, denoted by F_i (i=0,…, M-1), by using their ordering in the permutated index sequence; (2) loads a certain number of vectors from each shuffled file, concatenates them sequentially in blocks, and save the concatenated result into an intermediate file, denoted by I_m (m=0,…,M-1); (3) shuffle each intermediate file according to the related proportion in the whole permutated index numbers. It should be noted that the second step will be repeated for each intermediate file. This procedure avoids frequent accesses to the disc as much as possible and considerably decreases the influence of the limited disc I/O bandwidth on the total permutation.

6.3 AUTO-ENCODER

In this project, a deep auto-encoder was constructed by stacking several auto-encoders. This network was first pre-trained on input image patches in a layer-wise, unsupervised manner. Afterwards, the whole network was fine-tuned using the standard back-propagation scheme for training a network which might be used for a classification task later. There will represent an opportunity to integrate a deep auto-encoder in order to replace hand-crafted pre-processing and classical learning in the first stages of some real-world applications.

In the generative pre-training and then the discriminative fine-tuning stages, the vectors in the training dataset are grouped into mini-batches, each of which has 128 training data vectors. The intuition is explained as follows. If the batch size is too small, it is inefficient to execute dense matrix operations, e.g. matrix-matrix multiplications on the GPU. On the other hand, a batch size which is too large often makes the training unstable.

In training the auto-encoder, the RBM training is where the vast majority of the computation takes place. In the procedure of training an RBM, samples from the training dataset are input to this RBM model through visible neurons. The network alternatively samples back and forth between the visible and hidden neurons. The model will learn the visible-hidden connection weights and neuron activation biases after dozens of training epochs, and then it learns how to reconstruct the input data during the phase where it samples the visible neurons from the hidden neurons.

7. A DEMO – A CONTEXT-BASD IMAGE RETRIEVAL SYSTEM

In this section, a demo will be built up in order to illustrate the function of the RBM and auto-encoder models as well as their performance. This section begins with a brief description of this demo; and then gives the computing platform; and finally some important issues related to the OpenCL implementation.

7.1 GENERAL DESCRIPTION OF THE DEMO

Automatic classification of images has been one of the challenges in visual information indexing and retrieval systems. The outcome of so far research efforts has been confined to specialized systems mostly based on the analysis of low-level image primitives. Relying on low-level features only, it is possible to automatically extract important relationships between images. However, such an approach lacks the potential to achieve accurate semantic-based image classification for generic automatic annotation and retrieval.

Knowledge- and feature-based clustering as well as topology representation are important aspects for the improvement of classification and annotation performance. The clustering of visual information unit can be based on the high-level top-down visual information perception and classification model. Its implementation uses a generative approach. This kind of hybrid structure can play a key role in modeling an autonomous visual information classification and retrieval system.

7.2 THE CONFIGURATION OF THE COMPUTING PLATFORM

The host is the gate keeper for the GPU devices. Its main functionality is to receive commands from the outside world, and translates them into internal commands for the rest of the pipeline.

7.3 ALGORITHM IMPLEMENTATION

The auto-encoder had the following architecture 336-1024-512-256, and it was trained in the way of using RBM models for the pre-training and the auto-encoder for the fine-tuning. The RBM models were initialized with very small random weights and zero biases and trained for 80 epochs using mini-batches of the size 128. For the linear-binary RBM, the learning rate is set equal to 0.001 on the average per-case gradient and for the binary-binary RBM, the learning rate is 0.01. At the beginning and the end of learning, the learning rate was reduced to prevent the gradient having big values and to minimize fluctuations in the final weights, respectively. The moment is set to 0.9 to speed learning.

The auto-encoder was initialized with the weights from the separately trained stacks of RBM models and then fine-tuned using the back-propagation method to minimize the root mean squared reconstruction error. In the auto-encoder, the hidden layers halve in size until it reached the desired size, i.e. 256. To force the codes to be binary, the outputs of the logistic units in the central layer were rounded to 1 or 0 during the forward-pass but ignored this rounding during the backward pass. For the auto-encoder, the learning rate is set equal to be 10-6 for all layers and the network was trained for 5 epochs. According to the tests made by the authors in [], the weights of the auto-encoder only changed very slightly from those weights derived by the RBM models, but the image reconstruction results have significantly improved quality.

The RBM and auto-encoder models were tested by using an image database, CIFAR, which has 80 million tiny low-resolution RGB color images over 79,000 non-abstract English nouns and noun phrases in the WordNet. These images have the size of 32 by 32 pixels. The images were collected by searching the Web with search engines (such as Google and Flickr) over the span of six months. Although some manual curation was conducted to remove duplicates and low-quality images, each image in the dataset has a noisy label or image labels are extremely unreliable because of search technologies.

Following the description made by Alex Krizhevsky and G.E. Hinton, an unlabeled subset of 1.6 million images were used for both training and retrieval. This subset contains only the first 30 images found for each English search term. After the

pre-training, the auto-encoder are fine-tuned using the query images from the CIFAR-10, which is a carefully labeled subset of the 80 million tiny images, containing 60,000 images split equally between ten classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each image in CIFAR-10 has been selected to contain one dominant object of the appropriate class and only 3% of the CIFAR-10 images are in the set of 1.6 million training images.

7.1 PRE-PROCESSING

Each image in the training set is partitioned into 81 overlapping patches, each of which has 16 by 16 pixels and is compressed into the "retina" format. According to this format, boundary pixels of a patch image are down-sampled at lower resolution, while the middle 8 by 8 pixels are kept as originally. Using a stride of 2 pixels, totally there are 81 distinct patches that can be extracted from a 32 by 32 image. Thereafter, patch-based visual features are extracted by training the auto-encoder on local patch images. The main reason for using a bag of patch images is that code words obtained at the entire image level are considered stable for capturing the global image structure and thus for finding similar images at this level, although the code words are not invariant to transformations like translation of one object within the image which often have little effect on the semantic content. The primary reason for using the variable-resolution "retina" format is to reduce the influence of the boundary pixels to the learning results.

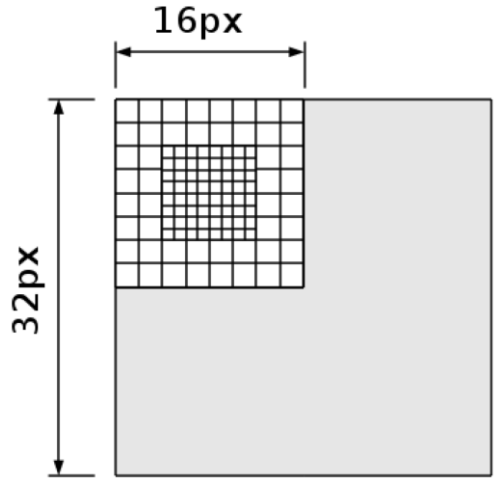

Fig. the variable-resolution retina format used to sample the 32 by 32 images.

After the retina reformatting step, an image becomes 81 vectors, each of which has 336 elements. Each vector is pre-processed by subtracting from each element its mean value over all vectors and then dividing by the standard deviation over all vectors.

7.2 SEARCHING

The procedure of searching images followed the original description presented in [].

1. Construct a 2^{28}-element array (the semantic hashing table) that stores at index i the list of images that have image patches with code i (possibly with repetitions, as one image may have multiple patches with the same code).
2. For each of the 81 patches of the query image, use the semantic hashing table to quickly find the set of images that have patches with codes no more than 3 bits away from the query patch code. Assign to each found image the score 2^{3-d}, where d is the hamming distance of the query patch code from the found image patch code. The hamming ball around the query patch code was explored in order of increasing distance, so each found image was given the maximal score it could get for a single patch. It does not get additional score if the same query

patch matches some other part of the same image equally or less well. This avoids such problems as one query patch of sky matching a large number of other patches of sky in another image.

3. Sum the scores for each image over all 81 patches of the query image and return the images in the order of descending score.
4. Finally, combine the ranked list of images found in the second step with the ranked list of images found using 256-bit deep codes by adding the two ranks of each image and re-ordering them.

CONCLUSIONS

In this project, two commonly used neural networks, i.e. RBM and auto-encoder, have been implemented in OpenCL on heterogeneous platforms. The OpenCL implementation contains the GPU code as well as the CPU code. The preliminary performance experiment results show that these networks are suitable for the GPU devices and even heterogeneous systems with because such platform can provide the mechanism of data communication between the CPU and the GPU devices, except the powerful computing resources in the GPU devices.

REFERENCES