Neural Networks Exam Project

Gabriele Angeletti

September 22, 2015

Theoretical study and Implementation of Restricted Boltzmann Machines and Deep Belief Networks

Introduction

In this project I implemented Restricted Boltzmann Machines with various types of units, and also a Deep Belief Network, made by a stack of RBMs. I tested the algorithms on the MNIST handwritten digits dataset[1].

Boltzmann Machines are energy models that learns to represent their own input distribution. They are made of two sets of units: the first set is made by the observable units, the ones that represent the input to the BM, the second set is made by hidden units, that learns a representation of the input distribution. In generic BMs every kind of connection is allowed: visible $<->$ hidden, hidden $<->$ hidden, visible $<->$ visible. In order to make learning and inference tractable, a new model with constrained type of connections was designed, the Restricted Boltzmann Machine (the only type of connection allowed is the visible $<->$ hidden type). Learning is done by minimizing an Energy function $E(v,h)$, that is a function of both the visible and hidden units. The main idea is that, given a visible configuration, configurations of the hidden units that are more probable under the model should have a lower energy.

$$P(v) = \sum_h P(v,h) = \sum_h \frac{e^{-E(v,h)}}{Z} \tag{1}$$

The problem is that the partition function used to normalize probabilities must be computed for each possible configuration of the visible and the hidden units, thus it is intractable. Contrastive Divergence learning (Hinton) works by approximating the data log-likelihood gradient under the assumptions that even if we don't know the posterior distribution in closed form, we can get unbiased samples from it using a Markov Chain Monte Carlo method. Contrastive Divergence maximizes the log probability of the data by "put-down" the energy of likely configurations and "raising-up" the energy of others consideration. Even if the theory requires unbiased samples from the Markov Chain (which can be very expensive), in practice even one step of sampling works well (the implementation works with an arbitrary number of samples, but I almost always used one/three samples in the experiments).

RBMs can have different type of visible and hidden units. In this project I implemented three types of RBM:

- RBM with stochastic binary units on both the visible and the hidden layers. A stochastic binary unit is a binary neuron that is on/off based on a probability defined by the Energy function.
- RBM with Gaussian units on the visible layer and stochastic binary units on the hidden layer (Gaussian-Bernoulli RBM). In many domains, like image recognition, binary units are a poor representation of the input. One would have real-valued units that better represents and preserves information about the input. This is achieved by having Gaussian visible units. The energy function is changed and the conditional probability of each visible unit become a Gaussian, with independent variance (the variance can be fixed of can be learned as well).
- RBM with multinomial units on both the visible and the hidden layers. Multinomial units can take discrete values and they are implemented as a simple variation of stochastic binary units. For example, if a state variable can take K different values, it is replaced by K binary stochastic units for which a 1-of-K constraint is imposed. In this way the learning rule is the same as RBM with binary units.

RBM with stochastic binary units

For this type of RBMs, the energy function to be minimized is:

$$E(v,h) = -b'v - c'h - h'Wv \tag{2}$$

Where b is the vector of biases of the visible units, c are the biases for the hidden units and W is the weight matrix. Because of the constraints imposed on the connections, visible and hidden units are conditional independent given one another. Because of this the posterior distributions can be written as a product of distribution for each unit (product of experts). Considering a single visible (hidden) unit, and substituting the energy function in (1), we have:

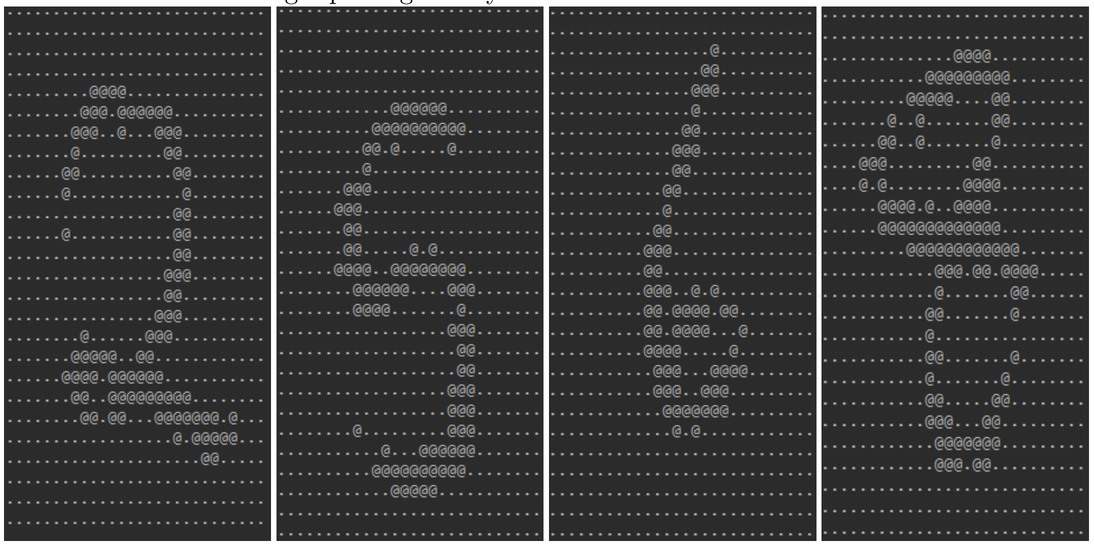
$$\begin{gathered} P(h_j = 1|v) = \sigma(c_j + W_j v) \\ P(v_k = 1|h) = \sigma(b_k + W'_k h) \end{gathered} \tag{3}$$

Where σ is the sigmoid function $\sigma(x) = 1/(1+e^{-x})$. The conditional probabilities for the entire visible (hidden) vector is found by: $P(v|h) = \prod_k P(v_k = 1|h)$, $P(h|v) = \prod_j P(h_j = 1|v)$. Contrastive Divergence learning approximate the log probability gradient using alternating gibbs sampling. The gradient assume a very convenient form:

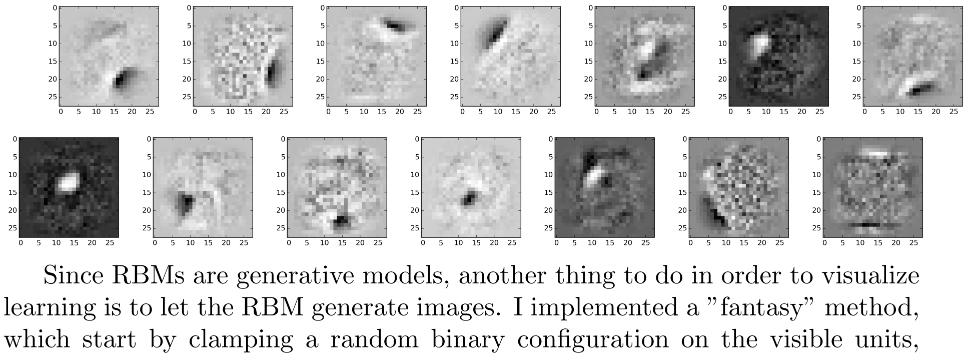
$$\frac{\partial log(P(v^0))}{\partial w_{j,k}} =< v_k^0 h_j^0 > - < v_k^\infty h_j^\infty > \tag{4}$$

The idea is that maximizing the log probability of the data is the equivalent of minimizing the Kullback-Leibler divergence, $KL(P^0||P_\theta^\infty)$, where P^0 is the distribution of the input data and $P_\theta^\infty)$ is the distribution defined by the model under parameters θ. The sampling works by clamping a sample from the

training distribution on the visible units, then compute the posterior over the hidden units, then reconstruct the visible units states from the posterior just calculated, and then recompute the posterior using the reconstruction. This is one complete step of Gibbs-sampling. As said before, despite the theory requires that the Markov Chain settles to equilibrium, in practice even only one step of Gibbs-sampling works very well.
When the RBM has been trained, it has learned a higher level representation of the input with respect of the raw input. One can then for example use a Logistic Regression layer over the higher level representation of the RBM in order to perform classification tasks. Using MNIST, I tested a RBM with a layer of Logistic Regression and confronted it with a Logistic Regression classifier trained directly from the raw input. In all the experiments I have made, RBM with LR greatly outperforms standard LR, sometimes with a variance of about 10% on the test set accuracy.
In order to see if the RBM was learning well during training, reconstructed samples under the model where printed after each iteration. After enough iteration, the RBM was reconstructing input digit fairly well.



In order to visually see what the RBM has learned on MNIST, I took the 784 weights from some hidden units to the visible units, and rearranged them to 28x28 images. One can visually see that each feature detector (hidden unit) has learned to filter the input image in some way (edges, circles, etc..)



Since RBMs are generative models, another thing to do in order to visualize learning is to let the RBM generate images. I implemented a "fantasy" method, which start by clamping a random binary configuration on the visible units, and then by performing alternating Gibbs-sampling steps until it reaches the equilibrium distribution. Then, one can get unbiased sample from the posterior distribution. By starting with a random initial configuration and by performing enough sampling steps (e.g. 1000), the RBM can generate digit images.

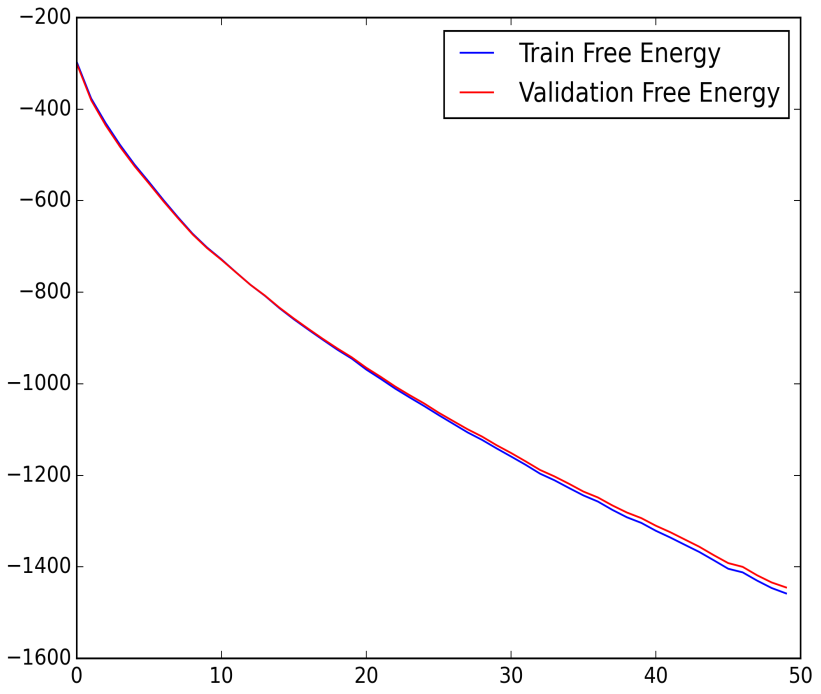
In order to debug RBM learning and to see when to stop one can use free energy. Free energy is defined from equation (1) as:

$$F(v) = -log \sum_{h} e^{-E(v,h)} \tag{5}$$

Which, for binary stochastic units becomes:

$$F(v) = -b'v - \sum_{j} log \sum_{hj} e^{h_j(c_j + W_j v)} \tag{6}$$

The free energy should be decreasing during learning. I computed two average free energies over each batch: the training set free energy and the validation set free energy. The validation energy is useful in order to see when to stop learning: if the train free energy is decreasing and the validation free energy starts increasing, then its time to stop learning. The following is a plot of train and validation sets free energies, for a RBM that was trained for 400 epochs using half of the full MNIST training set (30k images), with 784 visible units and 500 hidden units. This trained RBM is the same from where I took the images of the weights above, and the same that I used as first RBM to fine-tune the Deep Belief Net. The plot clearly shows that the RBM could have been trained for more epochs, since both the free energies were decreasing.



RBM with Gaussian visible units

When using the MNIST dataset to learn a binary RBM, I first normalized the pixel intensities to be in the range [0,1] and then used this values as probabilities that a pixel is inked or not. This way I got a "binary version" of MNIST. For more complex image recognition tasks, logistic units are a poor representation of the input. Representing pixel intensities as real values would be a better way of retaining the information in the image. RBM with Gaussian units are an extension that works with real valued units. In this project only the visible units are Gaussian, the hidden units are binary and works like in the previous type of rbm.
For the Gaussian units, the energy function is different, and so is different the expression to compute the conditional probabilities. The energy function is:

$$E(v,h) = \frac{\|v-b\|^2}{2\sigma^2} - c'h - \frac{v'Wh}{\sigma^2} \tag{7}$$

With this energy function, the visible units given the hidden units are Gaussian distributed with standard deviation σ. The standard deviation σ can be fixed for all units or one σ per unit can be learned. In the implementation I used a fixed sigma for all the units. Under this energy function, the conditional probabilities becomes:

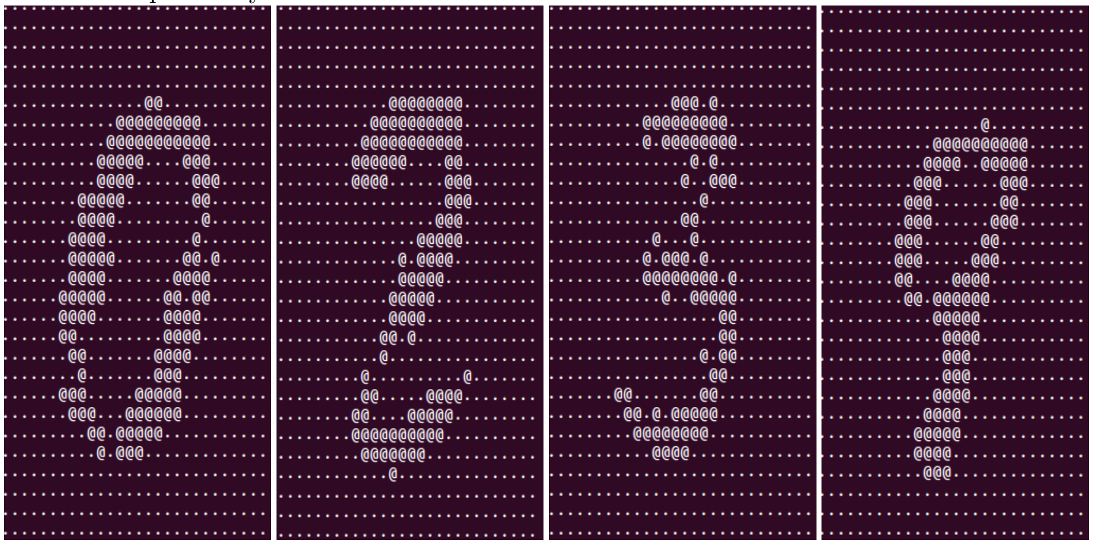
$$\begin{aligned} P(h_j = 1|v) &= \sigma(c_j + \frac{W_j v}{\sigma^2}) \\ P(v|h) &= N(v; b + Wh, \sigma^2) \end{aligned} \tag{8}$$

The hidden conditional probabilities are the same as before, with the only difference of the variance dividing the dot-product between the weights and the visible. The visible conditional probability is a Gaussian with variance σ^2 and mean $b + Wh$. Like before, the log-likelihood gradient is given by the difference between the expectations of the energies gradient under the data and the model distribution, and again we approximate the model distribution by performing a finite number of Gibbs-sampling steps (Contrastive Divergence). The learning rule for weights and biases in this case are:

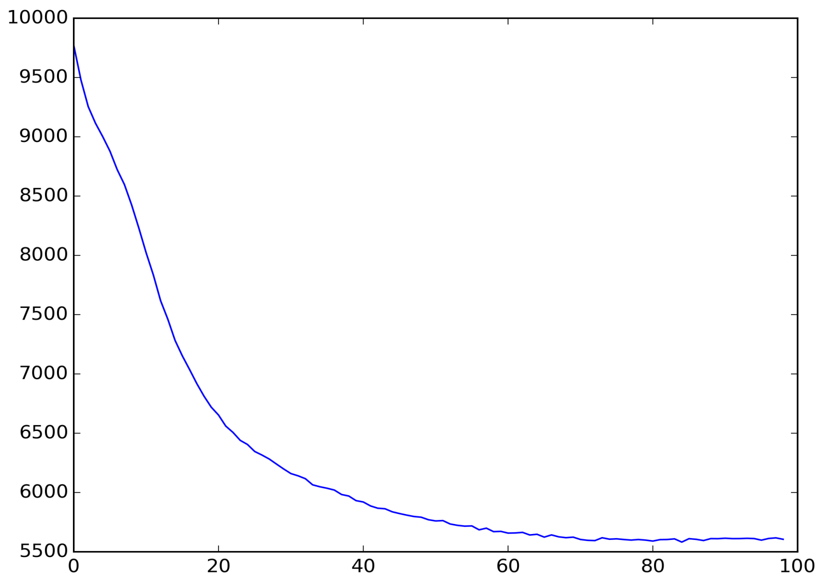
$$\begin{aligned} \partial vbias &= \frac{data - vbias}{\sigma^2} - \frac{modeldata - vbias}{\sigma^2} \\ \partial hbias &= P(h = 1|data) - P(h = 1|modeldata) \\ \partial w = \frac{dataP(h = 1|data)'}{\sigma^2} &- \frac{modeldataP(h = 1|modeldata)'}{\sigma^2} \end{aligned} \tag{9}$$

Where modeldata is the reconstruction of the data under the model after k steps of Gibbs-sampling.
The less binary units can represent well the input distribution, the more Gaussian RBM will be better than binary RBMs at representing the input distribution. For MNIST Gaussian RBM slightly outperform binary ones, and surely the gap would be more high for more complex tasks like face recognition or image-net.
The following are sampled reconstructed on the Gaussian visible units during

learning. In my experiments, on average, the Gaussian RBM need a few more training epochs before start printing digits then binary RBM, but they both reconstruct samples fairly well.



I trained a Gaussian RBM for 100 epochs using only 7k images, with 784 visible units and 250 hidden units. Then I put a Logistic Regression layer over the G-RBM, and it achieved an accuracy of 90.75% on the test set. A binary RBM with the same characteristics (and a learning rate 20 times higher), achieved an accuracy of 87.84% on the test set. Since Gaussian units can take arbitrary real values, they are much less stable and thus more difficult to train than binary units. Hence they need a much lower learning rate. Below there is a plot of the mean square error of the Gaussian RBM during training.

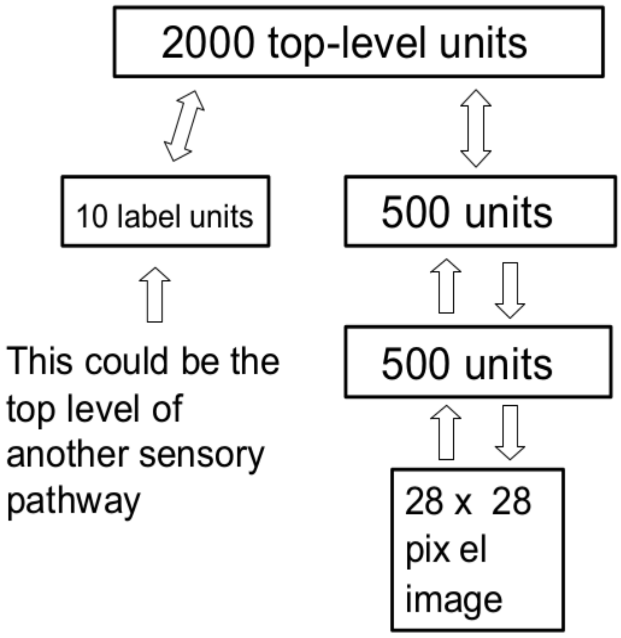


RBM with multinomial units

RBMs with multinomial units are almost the same as binary RBMs. If we have for example 10 visible units and each unit can take the values $\{1,2,3,4,5,6\}$, we represent each of the 10 units with 6 binary units and we impose the constraint that for each of this groups, only one-out-of-6 binary units can be on at each moment. The RBM will have 60 visible units, and the hidden units can be multinomial as well or stochastic binary. Using this representation of discrete states the learning rule is exactly the same as for binary RBMs, with the only difference that one must ensure the one-out-of-k constraints.

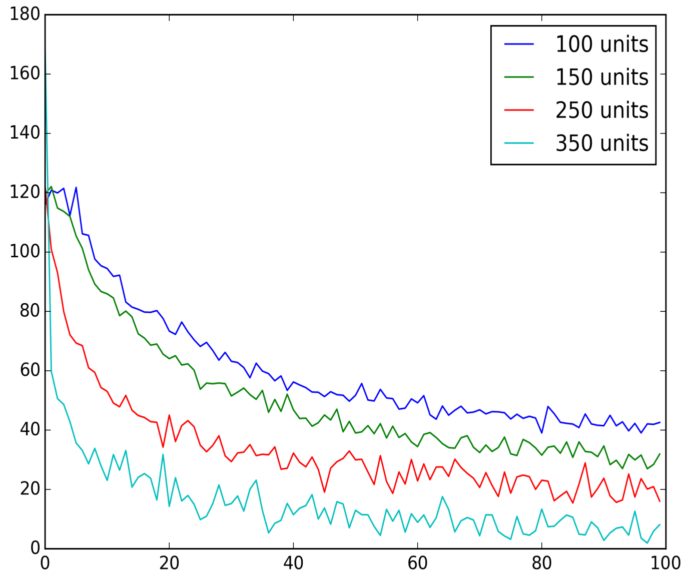
Deep Belief Networks

Restricted Boltzmann Machines has one layer of hidden units, that learns a higher level representation of the input. If we take this higher level features and use them as input for a second RBM, this second RBM will learn higher level features with respect to the higher level features learned by the first RBM. This way we can construct hierarchical representations of features, where each level in the hierarchy learn higher level features with respect to the layer below it. This type of training is called unsupervised greedy layer-wise pre-training of deep nets.
Deep Networks were always been difficult to train, because of the vanishing gradients (explaining away) problem. The idea is that deep nets has lots of hidden layers, but because the errors are defined only in terms of the output layer, when back-propagating the errors after few layers there is no useful information to be propagated backwards. Unsupervised pre-training can be viewed as a way of solving this problem by sensibly initialize the weights of the deep net, so that successive fine-tuning can find a good local optima. In the project the Deep Belief Net is implemented using only stochastic binary RBMs. I think that using a Gaussian RBM to model the input distribution and then binary RBMs after would have lead to better results, but it would have taken lot of training time. The architecture I used is the same used by Hinton et. al 2006 [2], and is depicted below (the image is taken from Hinton's paper):



A Deep Belief Network is a generative model, with the lower layers that form a directed Sigmoid Belief Network (the direction is top-down). The bottom-up connections are used for recognition and they are not part of the generative model. In the top layer RBM (in this case the one made up by 500 + 10 visible units and 2000 hidden units) the connections are undirected and it forms the top level associative memory of the net.
For supervised fine-tuning of the net I implemented the variation of the wake-sleep algorithm proposed by [2]. The algorithm basically works in a two step process: during the wake phase, a data vector is clamped on the visible units of the first RBM, and a bottom-up pass is performed using the bottom-up recognition connections, until the states of all RBM (except the top level) are defined. Then, the label corresponding to the data vector is clamped on the label units, and Gibbs-sampling is performed on the top level associative memory to get a sample of the labels and the last layer under the model.
In the sleep phase a top-down generative pass is performed to get samples from the model, until the visible layer of the first RBM. Then all the parameters of the model are updated.

In order to learn and fine-tune a DBN, I first trained one RBM with units 784x500 (the same from which I took the plots and images above), and then I trained another RBM with units 500x500. I used the first RBM to sample the dataset, and then the resulting hidden states were used as training set for the second RBM. After this unsupervised pre-training step, I ran the wake-sleep algorithm in order to fine tune the network. In Hinton's et al. architecture they used 2000 top-level units for the associative memory. For hardware and training-time reasons I couldn't use such a great number of units, so the results for the DBN where good enough to outperform Logistic Regression, but not good enough to surpass the single RBM with Logistic Regression. My hypothesis for this is that more training and a higher number of top level units is required in order to learn a very good model with the DBN. Also the lower level RBMs could have been trained more, as one can see from the train and validation free energies. In order to have a very good weights initialization for fine-tuning, the RBM should have been trained more. Below there are plots of the cross-entropy error during the wake-sleep algorithm for different number of top-level units (the number of units is much less than the 2000 used by [2], and also the number of images used is much much less). The plot shows that the more top-level units are used, the less the cross-entropy error.



After fine-tuning the dbn I used it to predict labels of images in the test set. The algorithm used to predict the label is pretty much the same as the wake phase of the wake sleep algorithm. The test image is clamped on the visible units of the first RBM, and then a bottom-up pass is used until the layer before the top level. Then, an "almost random" configuration of the label units is picked-up (almost random because the probability of being on is 0.1 instead of 0.5, this is done in order to reduce the iterations of sampling needed to reach equilibrium). Then, Gibbs-sampling in the top level associative memory is performed (changing only the label units and keeping fixed the image units). After sampling, the top-level layer will generate the correct label for the image.

References

[1] The MNIST database of handwritten digits by: Yann Lecun, Corinna Cortes

[2] A fast learning algorithm for Deep Belief Nets - 2006. Hinton, Osindero and Yee-Whye Teh