10-708: Probabilistic Graphical Models 10-708, Spring 2018

20 : Introduction to Deep Learning

Lecturer: Kayhan Batmanghelich *Scribes: Amir Alavi, Yu Chen*

1 Background and Motivations

Neural network research, though has existed for decades, has recently taken off as a hot new field, especially with the rise of deep neural network architectures. This growth has coincided with the newfound capabilities made possible with faster hardware and large amounts of data, and has brought with it billions of investment dollars from companies such as Google's DeepMind. These new tricks include:

- Deeper architectures
- Better optimization techniques
- Rectified Linear Units
- General purpose GPU (GP-GPUs) for accelerated matrix operations

The appeal of deep learning is that it has been shown to be the state-of-the-art in many pattern recognition competitions (most famously in image recognition competitions) and has done so with minimal feature engineering. Whereas older computer vision required hand crafted features (e.g. SIFT or HOG), deep learning methods learn the best features for the objective at hand.

1.1 Basic Recipe

Like any other machine learning problem we have looked at, we have:

1. Training data: $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^N$
2. Choice of decision function: $\hat{\boldsymbol{y}} = f_\theta(\boldsymbol{x}_i)$
   - Linear regression
   - Logistic regression
   - Neural network
3. Choice of loss function: $l(\hat{\boldsymbol{y}}, \boldsymbol{y}_i) \in \mathbb{R}$
   - Mean-squared error
   - Cross entropy
4. Training objective: $\boldsymbol{\theta}^* = \arg\min_\theta \sum_{i=1}^N l(f_\theta(\boldsymbol{x}_i), \boldsymbol{y}_i)$
5. Train with SGD (take small steps opposite the gradient): $\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla l(f_\theta(\boldsymbol{x}_i), \boldsymbol{y}_i)$

Key idea: SGD requires computing the gradient of the objective function with respect to each of our parameters. Because neural networks employ differentiable functions throughout (or operate in differentiable regions of functions), then we can compute gradients using the Chain Rule. This is how backpropagation works.

2 Deep Neural Networks

2.1 Background: Decision Functions with Shallow Networks

The simplest neural network decision function is a one-layer regression model depicted in Figure 1.

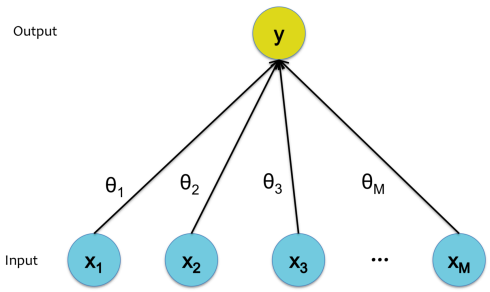

Figure 1: Linear regression layer where y is a scalar and $y = f_{\boldsymbol{\theta}}(\boldsymbol{x}) = h(\boldsymbol{\theta} \cdot \boldsymbol{x})$ and $h(a) = a$ (just a linear activation).

This boils down to learning a mapping from the M-dimensional input space to the 1-dimension output (label) space.

Logistic regression could be implemented as a neural network with the same architecture, but we would change the activation function to be the logit function (sigmoid activation) with $h(a) = \frac{1}{1+exp(a)}$. This could then be used for binary classification as you would with any logistic regression model.

2.2 Background: Multi-layer Neural Networks

Using the shallow networks defined in section 2.1 as a building block, we can build up a multi-layer neural network where the intermediate variables are the shallow modules we defined above. this is depicted in Figure 2, and can be extended to have many many hidden layers, where each hidden layer is viewed as a module (deep architectures).

The intuition is that as you go from the lower levels (input, pixels) to the higher levels (output, tag), the layers in between will be learning more and more abstract representations of the data (e.g. pixels → edges → eyebrows → face → human). This is a rough generalization and we don't actually know the "right" abstractions, but we let the model learn these.

The "learning" happens as before, using SGD, and utilizing backpropagation through the differentiable layers of the network as our method for calculating gradients (i.e. calculating "blame").

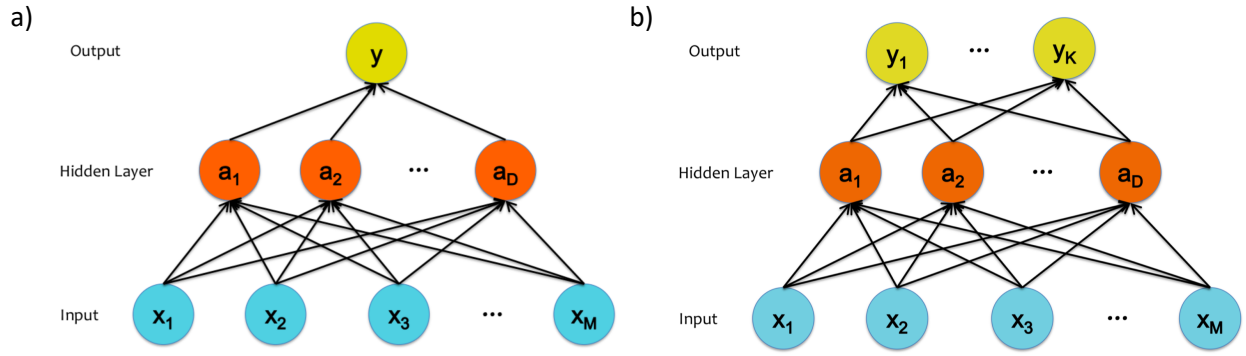

Figure 2: Multi-layer neural network with one hidden layer with a) one output b) multiple outputs. Hidden layer can be a linear or non-linear function of the input variables, and final output is a linear or non-linear function of the hidden layer.

2.3 Training a Deep Net

Using the MNIST data set and handwritten number recognition as our running example, we will examine different flavors of training the multi-layer neural networks introduced in section 2.2

2.3.1 No Pre-Training

"No pre-training" just means to initialize our parameters randomly, and then update them through SGD using backpropagation as mentioned in the basic recipe for neural network training in section 1.1.

The key ingredient here is **backpropagation**. For example, assume we have a simple neural network with input $\boldsymbol{x}$, a single hidden layer (intermediate variable) $\boldsymbol{u}$, and final output $\boldsymbol{y}$. Then we can formulate the forward propagation of our network as:

$$\boldsymbol{y} = g(\boldsymbol{u}) \text{ and } \boldsymbol{u} = h(\boldsymbol{x})$$

This is the basic module of forward propagation. Then to calculate the gradient of the ith output neuron with respect to kth input neuron, we can apply the Chain Rule as follows:

$$\frac{dy_i}{dx_k} = \sum_{j=1}^{J} \frac{dy_i}{du_j} \frac{du_j}{dx_k}, \quad \forall i, k$$

Where the sum is over all units in the intermediate layer (i.e. $\boldsymbol{u} \in \mathbb{R}^J$). **This is the basic module of backpropagation.** The forward and backward propagation for the simple case of logistic regression is depicted in Figure 3.

Adding more hidden layers as modules to the simple model from Figure 3 simply corresponds to *function composition*, which is unrolled with the same forward and backward propagation modules mentioned above, as expected.

The challenge around a decade ago was that we would have to do these calculations ourselves, by hand (write the code ourselves). Nowadays, there exist many general-purpose, automatic differentiation packages (e.g. PyTorch, Tensorflow, etc) that do it for us.

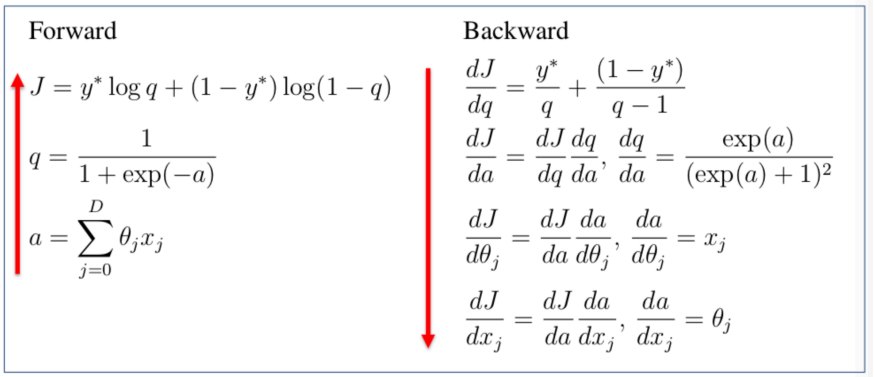

Figure 3: Forward and backward propagation for the simple logistic regression model described in Figure 1. Translating this notation to apply to Figure 1: J = Loss function on y, and $q = h$

Training a shallow neural network (i.e. logistic regression) following the basic recipe (random initialization, SGD + Backpropagation) would result in under 2% error (Figure 6). However, in the same figure we can see that training a deep neural network with the basic recipe would result in poorer performance of almost 2.5% error. This is because although the deeper architecture is more powerful and complicated (has larger capacity), it takes longer to train and likely has not converged, resulting in less than optimal parameters and poorer performance.

The issues with this approach is that the objective function we are trying to optimize is highly non-convex, meaning that we have many saddle points and local optimums that SGD can get stuck in. This is due to our random starting points being at bad location in the parameter space.

Another problem is known as the "vanishing gradient" problem. The chain rule multiplies many partial derivatives together. For the non-linearities typically used like $tanh(x)$ and $sigmoid(x)$, as these neurons saturate (move away from $x = 0$), the gradients become smaller and smaller, and small numbers multiplied together move very close to zero. This is why we say the gradients can "vanish".

How can we deal with these issues?

2.3.2 Supervised Pre-Training

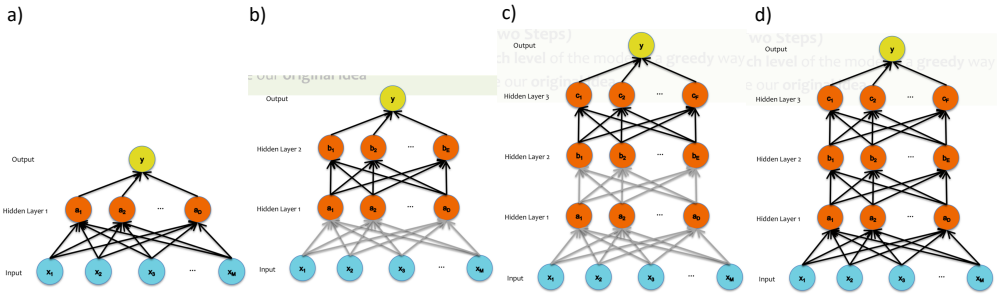

Figure 4: Layer-wise supervised pre-training of a multilayer neural network architecture. Dark arrows represent parameters that are being optimized. Grey arrows represent "frozen" parameters. a) Pretraining first hidden layer. b) Pretraining second hidden layer. c) Pretraining third hidden layer. d) Fine-tuning entire network using pretrained weights as initialization.

One solution is to stop using the purely random initializations, which tended to give us bad starting points. The idea here is to take each layer of our multi-layer neural network, and train it as its own single-hidden layer neural network. We start from the bottom of the network and work our way up to the last layer. After training the parameters of an individual layer, we fix them ("freeze" the weights) and continue training the next layer. **Note: the input to the next layer has to go through the layer below first, so we must propagate the input data from the very bottom through all of the frozen layers until we get its representation which will be the input to the current layer.** Once we have done this for all hidden layers, we take all of these "frozen" weights and set them as the initial weights for the overall multi-layer network. Then we can "unfreeze" them and train the entire network for a few iterations to "fine-tune" it. This entire process is depicted in Figure 4

The intuition here is that we are simply starting out with a better initialization of our parameters compared to the strategy in section 2.3.1. In other words, we are placing our initial parameter guesses in a better regime, closer to the optimal values and thus less likely to get stuck in sub-optimal regions.

With this strategy, the neural network improves over the deep net that had no pretraining, and achieves 2% error, but still does worse than the simple shallow net (Figure 6).

2.3.3 Unsupervised Pre-Training

Another strategy is to use an unsupervised strategy to pretrain each of the layers. Similar to the approach in section 2.3.2, this will also happen in a greedy, layer-wise fashion. However, now we will use Autoencoders as our "greedy" pretraining module rather than the classification module we used in 2.3.2.

Autoencoders are neural networks whose output dimensions are identical to the input dimensions, and whose intermediate layers usually have cardinality smaller than the input/output layers. Thus, an autoencoder is trained with the objective of reconstructing the input data, but only after passing it through an under-complete intermediate "code" layer. The objective function can be mean-squared error. This basic architecture is depicted in Figure 5.

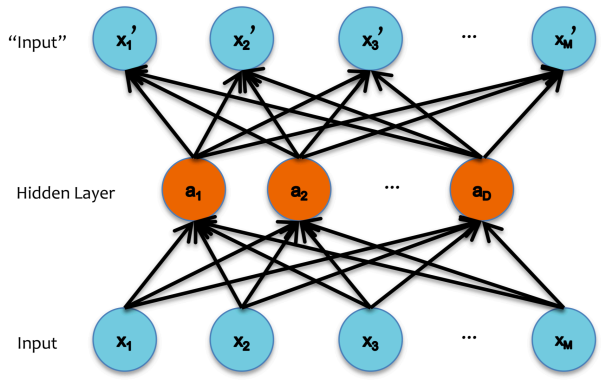

Figure 5: A basic autoencoder module architecture.

The intuition here is that we want to learn some sort of "abstraction" of the data that doesn't depend on the labels, but rather depends on the structure of the data itself, and that this might be a better initialization.

Finally, with this strategy, we are able to outperform the simple shallow network with a smaller error of less than 1.5%. This shows that, yes, deeper architectures can outperform shallow architectures, but with the caveat that you need the right initialization. Nowadays however, with access to fast GPUs, we don't need to do any pretraining, we can run for more iterations.

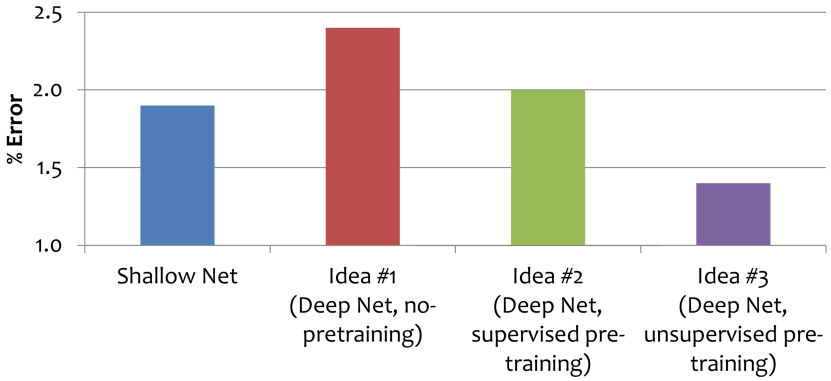

Figure 6: Performance of various neural network architectures and training strategies on the MNIST data set.

3 Deep Belief Networks

So how does deep learning relate to graphical models?...The original Deep Learning papers circa 2006 were concerned with a flavor of neural networks that happened to also be a flavor of Belief Networks.

3.1 Sigmoid Belief Network

The basic structure of a sigmoid belief network is depicted in Figure 7. Here we see that each of our pixels in our image represents a random variable in the observed or "visible" layer, and we have a set of hidden layers representing latent random variables. All random variables are binary. You can think of the joint distribution as a bunch of sigmoids multiplied by each other.

$$p(x_i|\text{parents}(x_i)) = \frac{1}{1 + exp(-\sum_j w_{ij}x_j)}$$

So we have the structure of a directed graphical model, but we do not know the weights W_1 and W_2 and must learn them.

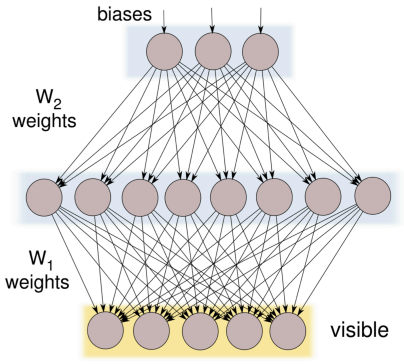

Figure 7: The basic architecture of a sigmoid belief network. Note that the arrows here correspond to directed edges between random variables in a PGM, not the direction of information propagation of a neural network.

3.1.1 Training via Contrastive Divergence

Contrastive Divergence is a general tool for learning a generative distribution, where the derivative of the log partition function is intractable to compute. For example, maximizing the likelihood procedure is,

$$\max_w l(\mathcal{D}, w) = \sum_{v \in \mathcal{D}} \log P_w(v) = \sum_{v \in \mathcal{D}} \log \frac{P_w^*(v)}{Z(w)} \tag{1}$$

where w is the parameter. $P_w^*(v)$ is the non-normalized distribution, and $Z(w)$ is the normalizer. In practice, the normalizer $Z(w)$ is usually intractable. The math trick to calculate the derivative of $\log Z(w)$ is the following,

$$\begin{aligned} \frac{\partial \log Z(w)}{\partial w} &= \frac{1}{Z} \frac{\partial Z(w)}{\partial w} \\ &= \frac{1}{Z} \frac{\partial}{\partial w} \int P^*(x) dx \\ &= \frac{1}{Z} \int \frac{\partial}{\partial w} P^*(x) dx \end{aligned} \tag{2}$$

The trick here is commonly seen that we multiple and divide the $P^*(x)$. Then,

$$\begin{aligned} & \frac{1}{Z} \int \frac{\partial}{\partial w} P^*(x) dx \\ =& \frac{1}{Z} \int \frac{\frac{\partial}{\partial w} P^*(x)}{P^*(x)} P^*(x) dx \\ =& \int \Big[\frac{\partial}{\partial w} \log P^*(x) \Big] \frac{P^*(x)}{Z} dx \\ =& \mathbb{E}\Big[\frac{\partial}{\partial w} \log P^*(x) \Big] \end{aligned} \tag{3}$$

However, the computation of the expectation over $P^*(x)$ requires lots of MCMC sampling. As Hinton suggested, in the beginning, we can only take few MCMC samples to approximate the gradient, thus make the computation faster. An other way to view this is to introduce hidden variables,

$$\frac{\partial}{\partial w} \log L = \frac{1}{N} \sum_{v \in \mathcal{D}} \sum_h P(h|v) \frac{\partial}{\partial w} \log P^*(x) - \sum_{v,h} P(v,h) \log P^*(x) \tag{4}$$

The first term is the expectation over conditional distribution $P(h|v)$, the second term is the expectation over the joint distribution of h, v. This procedure is called wake-sleep. In the wake phase, the model is exploring the model space, and in the sleep phase, it figures out the landscape of the data.

3.1.2 Back to Sigmoid BN

If we apply the above wake-sleep theory to the Sigmoid BN, the second term (sleep phase) is zero, because the model is Bayesian network and the normalizer is 1. The first term is,

$$\frac{\partial}{\partial w_{ij}} \log L = (x_i - p_i) x_j \tag{5}$$

This method uses stochastic gradient ascent in the M-step of EM algorithm.

NOTE: It is worth noting that the Sigmoid BN is easy to do sampling, but it is hard to learn.

3.2 Boltzman Machines

3.2.1 Basic concepts of Boltzman machines

Boltzman machines are popular undirected graphs, where the potentials are defined in a parametric way, where the energy function is in a quadratic format,

$$\Phi_{ij}(x_i, x_j) = \exp\left\{x_i W_{ij} x_j\right\} \tag{6}$$

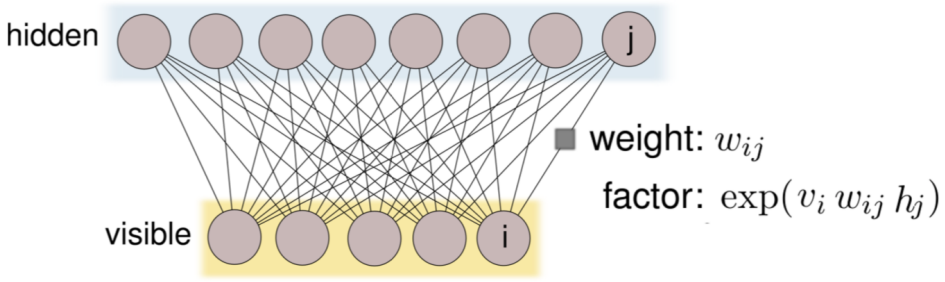

Figure 8: An example layout of undirected restricted Boltzman machine.

For *restricted Boltzman machines*, visible units are one layer, and hidden units are another. There are no inner connections within hidden layer nor visible layer. Edges only connect across layers, as shwon in Figure 11. Because of this 'restricted' property, v_i and v_j are independent given all hidden variables. Since none of the units within a layer are interconnected, we can do Gibbs sampling by updating the whole layer at a time.

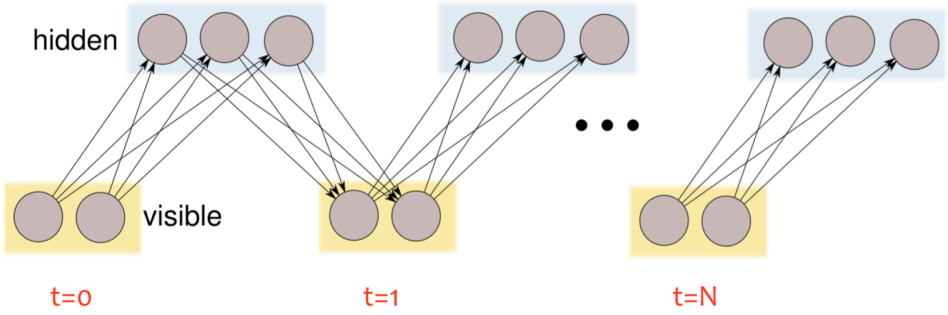

Figure 9: An example layout of undirected restricted Boltzman machine.

3.2.2 Learning in restricted Boltzman machines

If we apply the contrastive divergence (wake-sleep algorithm) here for restricted Boltzman machine, we get the gradient,

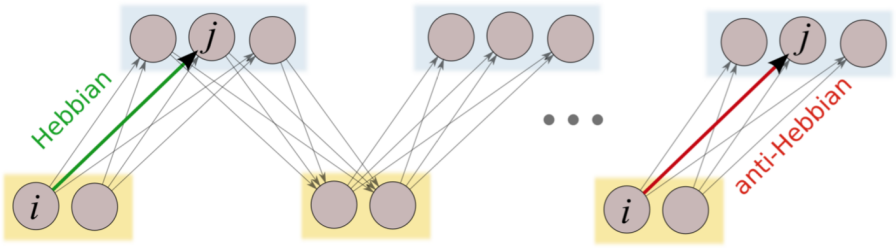

Figure 10: An example learning the undirected restricted Boltzman machine.

$$\frac{\partial}{\partial w_{ij}} \log L = \mathop{\mathbb{E}}_{v \in \mathcal{D}, h \sim P(h|v)} [x_i x_j] - \mathop{\mathbb{E}}_{x \sim P(x)} [x_i x_j] \tag{7}$$

The first term is corresponding to the wake phase, and in the context of restricted Boltzman machine, it is also called Hebbian. The second term corresponds to the sleep phase, called anti-Hebbian.

3.2.3 Learning in standard Boltzman machines

In a standard Boltzman machine, in the visible layer or hidden layer, nodes are allowed to connected to each other. The density is defined as,

$$p(\mathbf{v}, \mathbf{h}) = \exp\left\{\frac{1}{2}\mathbf{v}^T L \mathbf{v} + \frac{1}{2}\mathbf{h}^T J \mathbf{h} + \mathbf{v}^T W \mathbf{h}\right\} \tag{8}$$

The model can be learned by 1. Approximate the data distribution by variational inference. 2. Approximate the model distribution with a persistent Markov chain (from iteration to iteration). When update the parameters,

$$\Delta L/\alpha = \mathop{\mathbb{E}}_{\mathbf{v} \in \mathcal{D}, \mathbf{h} \sim P(\mathbf{h}|\mathbf{v})} [\mathbf{v}\mathbf{v}^T] - \mathop{\mathbb{E}}_{\mathbf{v}, \mathbf{h} \sim P(\mathbf{v}, \mathbf{h})} [\mathbf{v}\mathbf{v}^T] \tag{9}$$

$$\Delta J/\alpha = \mathop{\mathbb{E}}_{\mathbf{v} \in \mathcal{D}, \mathbf{h} \sim P(\mathbf{h}|\mathbf{v})} [\mathbf{h}\mathbf{h}^T] - \mathop{\mathbb{E}}_{\mathbf{v}, \mathbf{h} \sim P(\mathbf{v}, \mathbf{h})} [\mathbf{h}\mathbf{h}^T] \tag{10}$$

$$\Delta W/\alpha = \mathop{\mathbb{E}}_{\mathbf{v} \in \mathcal{D}, \mathbf{h} \sim P(\mathbf{h}|\mathbf{v})} [\mathbf{h}\mathbf{v}^T] - \mathop{\mathbb{E}}_{\mathbf{v}, \mathbf{h} \sim P(\mathbf{v}, \mathbf{h})} [\mathbf{h}\mathbf{v}^T] \tag{11}$$

Where α is the learning rate.

3.3 Unsupervised Learning of DBNs

Unsupervised learning task can be achieved through autoencoder, which is composed of pre-train a stack of RBMs in greedy layerwise fashion. We can unroll the restricted Boltzman machines to create an autoencoder. Finally, once we train the each layer using just inputs, we replace the output using observed labels. Then we fine-tune the parameters using backpropagation.

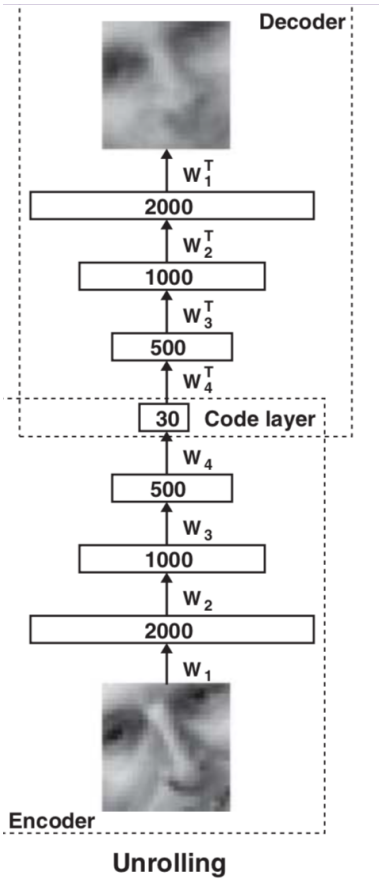

Figure 11: Autoencoder structure. The encoder decoder are stacked in a top-down and bottom-down fashion.