BUILDING HIGH LEVEL FEATURES USING UNSUPERVISED LEARNING

A Project Report Submitted

for the Course

MA498 Project I

by

Arvind Choudhary

(Roll No. 120123009)

to the

DEPARTMENT OF MATHEMATICS

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

GUWAHATI - 781039, INDIA

November 2015

CERTIFICATE

This is to certify that the work contained in this project report entitled "**Building high level features using unsupervised learning**" submitted by **Arvind Choudhary** (**Roll No.: 120123009**) to Department of Mathematics, Indian Institute of Technology Guwahati towards the requirement of the course **MA498 Project I** has been carried out by him under my supervision.

Guwahati - 781 039	(Dr. Arabin Kumar Dey)
November 2015	Project Supervisor

ABSTRACT

The main aim objectives of the project are:

Exploring and building features using Restricted Boltzmann Machine.

Contents

Chapter 1

Introduction

1.1 Artificial Neural Networks

Neural Networks takes a large number of training examples, and then develop a system which can learn from those training examples. By increasing the number of training examples, the networks learn more and improves the accuracy.

Along the way, we'll first define artificial neurons and next, learning algorithm for neural network called stochastic gradient descent and a fast way to compute gradient of the cost function (Backpropagation).

1.1.1 Artificial Neurons

Perceptron Neuron

A perceptron takes several binary inputs, x_1, x_2, x_3,.. and gives single binary output. Futher, every input x_i has a weight w_i expressing importance of the respective input. The neuron's output is determined by comparing the weighted sum $\sum_i w_i x_i$ with some threshold value of the neuron. In algebraic terms:

$$output = \begin{cases} 0 if \sum_i w_i x_i \leq threshold \\ 1 otherwise \end{cases}$$

Further simplifying, we write weighted sum $\Sigma_i w_i x_i$ as a dot product w · x and replace threshold with negative of bias, b. So, the perceptron rule can be rewritten as:

$$output = \begin{cases} 0 if w \cdot x + b \leq 0 \\ 1 if w \cdot x + b > 0 \end{cases}$$

The bias can be seen as measure of how easy it is to get perceptron to fire.

Sigmoid Neuron

Sigmoid neurons are similar to perceptrons, but modified so that small change in their weights and bias cause only slight revision in their output. So, instead of 0 or 1, the inputs can also take on any value between 0 and 1. The output is defined as $\sigma(w \cdot x + b)$, where σ is called sigmoid fuction and is defined as: $\sigma(x)$ = 1/(1+exp(-z))
Explicitly,

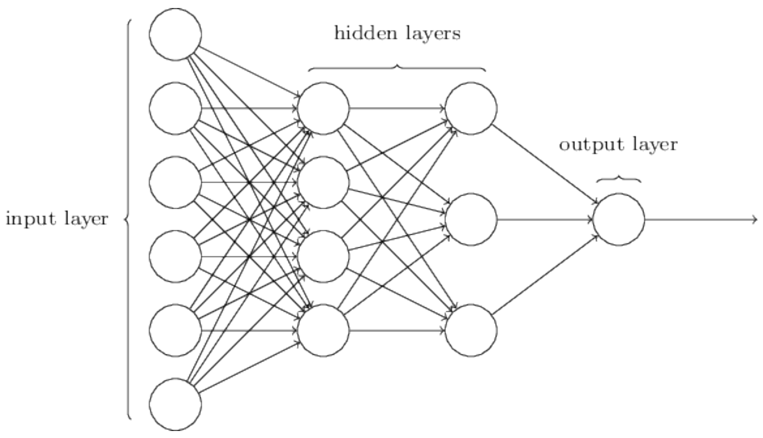
$$output = \frac{1}{1 + exp(-w \cdot x - b)} \tag{1.1}$$

The sigmoid function is chosen as the smoothness of σ means that small change in δw_j in weights and δb in bias will produce small change $\delta output$ from neuron.
From calculus:

$$\Delta \text{output} \approx \sum_j \frac{\partial \, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \, \text{output}}{\partial b} \Delta b, \tag{1.2}$$

The neuron gives a probable idea that a complex network of perceptrons could make quite accurate decisions which may consists of multiple layers of neurons. In this way, many layer network can make a sophisticated decision making.

1.1.2 Architecture



Such multiple later networks are called multilayer perceptrons or MLPs. As one layer is used as input to the next layer, such networks are called feedforward neural networks.

Chapter 2

Backpropagation

We use MNIST handwritten data as a training data set. We define our Objective / Cost function and minimise it by finding set of weights and biases using Gradient Descent. Next, we define Backpropagation algorithm which can efficiently compute gradients.

2.1 Learning with Gradient Descent

We need algorithm which lets us find weights and biases so that the output from the network approximates y(x) for all training inputs x. To determine how well we're achieving this goal, we define a Cost / Objective function:

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2. \tag{2.1}$$

- w is collection of all weights in the network

- b is collection of all biases

- n is total training inputs

- a is the vector of outputs from network when x is input.

2.2 Gradient Descent

To minimize cost by finding set of weights and biases, we use Gradient Descent. Let's suppose we're trying to minimize some function, C(v) with many variables v = v1, v2, v3,... For small Δv_2 in v_2 direction and Δv_1 in v_1 direction, we get:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2. \tag{2.2}$$

To make ΔC negative, we define
Δv as vector of changes in v, $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$ and
gradient of C as vector of partial derivates,

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T. \tag{2.3}$$

ΔC can be rewritten as

$$\Delta C \approx \nabla C \cdot \Delta v. \tag{2.4}$$

This equation basically helps us in selecting Δv so that we can find negative ΔC. Suppose we choose Δv as:

$$\Delta v = -\eta \nabla C \tag{2.5}$$

where η is a small positive parameter (learning rate). Above equation tells us that $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$ This guarantees that $\Delta C \leq 0$, i.e., C will always decrease, never increase.Hence,

$$v \rightarrow v' = v - \eta \nabla C. \tag{2.6}$$

2.2.1 Gradient Descent and Neural Network

We apply same gradient descent update rule in terms of components w_k and b_l, we have

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k} \tag{2.7}$$

$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l} \tag{2.8}$$

2.2.2 Stochastic Gradient Descent

The cost function has the form $C = \frac{1}{n} \sum_x C_x$ and averages over costs $C_x \equiv \frac{\|y(x)-a\|^2}{2}$ for every training example. In practice, we need to calculate gradients ∇C_x separately for each training example, x, and the take average. With the increase in training examples, this learning occurs slowly and there, we use Stochastic Gradient Descent.
To estimate ∇C, we randomly choose training inputs and calculate ∇C_x. It turns out to be a good estimate of true gradient ∇C and helps speed up gradient descent.
For small number m of randomly chosen training inputs, we label them as $X_1, X_2, ..., X_m$ and call them mini-batch. The average value of ∇C_{X_j} esti-

mates the average over all ∇C_x, i.e.,

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C, \tag{2.9}$$

or,

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j}, \tag{2.10}$$

Suppose w_k and b_l denote the weights and biases in our neural network. Then stochastic gradient descent works by picking out a randomly chosen mini-batch of training inputs, and training with those,

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{2.11}$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l}, \tag{2.12}$$

2.3 BackPropagation

Backpropagation is a general method to train artificial neural networks which is used along with gradient descent. Notations in feedforward neural network:
w^l_{jk} : weight from the k^{th} neuron in $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer.
b^l_j : bias of the j^{th} neuron in the l^{th} layer.
a^l_j : activation of the j^{th} neuron in the l^{th} layer.
z^l : weighted input.

We can relate activation of j^{th} neuron in l^{th} layer with $(l-1)^{th}$ layer as:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right), \tag{2.13}$$

or in vectorized form:

$$a^l = \sigma(w^l a^{l-1} + b^l) = \sigma(z^l) \tag{2.14}$$

We define weighted input as: $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$
or in vectorized compact form: $z^l = w^l a^{l-1} + b^l$

2.3.1 Assumptions about cost function

$$C = \frac{1}{2n}\sum_x \|y(x) - a^L(x)\|^2, \tag{2.15}$$

Assumption 1: Cost function C can be written as an average $C = \frac{1}{n}\sum_x C_x$ over cost functions C_x for training examples, x.
Assumption 2: Cost can be written as a function of the outputs from the neural network.
Quadratic cost function satisfies as:

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2, \tag{2.16}$$

2.3.2 Fundamental equations

Notation:

δ_j^l : Error in the j^{th} neuron in the l^{th} layer.

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}. \tag{2.17}$$

Equations

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{2.18}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{2.19}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{2.20}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{2.21}$$

Algorithm steps

1. **Input x**: Set the activation a^1
2. **Feedforward**: For l = 2,3,4,...,L, compute z^l and a^l
3. **Output error δ^L**: Compute vector δ^L
4. **Backpropagate the error**: For each l = L-1, L-2,...,2, compute δ^l
5. **Output**: The gradient of the cost is gives by $\frac{\partial C}{\partial b_j^l}$ and $\frac{\partial C}{\partial w_{jk}^l}$.

2.4 Implementation

We implement neural network to recognize handwritten digits. We use MNIST data set, which contains images of handwritten digits together with classification of digits. This training data consists of 28 by 28 grayscale images, so we set input layer to 784 neurons = 28 x 28. The pixel value varies between 0.0 to 1.0. We have 10 output neurons representing digits from 0 to 9 (all set to zero except one). The value is selected upon highest activation value.

Input: x as 784 dimensional vector.

Output: y = y(x) as 10 dimensional vector.

With Stochastic gradient descent, we randomly choose mini-batch of training examples. After, we pick another randomly chosen mini-batch, we continue until we exhaust the training inputs. This is called completing an *epoch* of training.

We further test our implementation with test images and find classification rate. `setup:`

η: `3.0`

`layers: 784,50,50,10`

`epoch: 20`

`mini-batch size: 10`

```
>>> net = network.Network([784,50,50,10])
>>> SGD(training_data, 20, 10, 3.0, test_data=test_data)
Epoch 0: 9083 / 10000
Epoch 1: 9315 / 10000
Epoch 2: 9346 / 10000
Epoch 3: 9423 / 10000
.
.
.
Epoch 17: 9593 / 10000
Epoch 18: 9585 / 10000
Epoch 19: 9594 / 10000
```

We see that trained network gives us classification rate of about 96 percent.

Chapter 3

Restricted Boltzmann Machine

A restricted Boltzmann machine (RBM) is a generative stochastic artificial neural network that can learn a probability distribution over its set of inputs. RBM is a Energy Bases Learning Model ans is a variant of Boltzmann machine, with the restriction that the neurons must form a bipartite graph: a pair of nodes from each of the two groups of units, commonly referred to as the "visible" and "hidden" units respectively, may have a symmetric connection between them, and there are no connections between nodes within a group. RBM performs Stochastic gradient descent to learn the weights and then applying W to the hidden layer gives the output which is considered as an approximation to the identity function.

3.1 Energy-Based Models (EBM)

Energy-based models associates a scalar energy to each configuration of the variables of interest. The configurations like to have low energy. Energy-based probabilistic models define probability distribution through an energy function as:

$$p(x) = \frac{e^{-E(x)}}{Z} \tag{3.1}$$

where Z is normalizing factor given by:

$$Z = \sum_x e^{-E(x)}$$

The idea is to minimize the negative empirical log-likelihood (or, maximize log-likelihood) of the energy function. Performming (stochastic) Gradient Descent, we estimate the parameters of model. We define the log-likelihood as:

$$L(\theta) = \frac{1}{N} \sum_x log(p(x))$$

and the loss function as negative log-likelihood.

$$l(\theta) = -L(\theta)$$

with gradient function as $-\frac{\partial log(p(x))}{\partial(\theta)}$ where, θ are the parameters of the model and summation runs over all the training samples.

3.1.1 EBMs with Hidden Units

If we have hidden part h, then the associated Energy function is $E(x, h)$ with x as visible unit. So,

$$P(x) = \sum_h \frac{e^{E(x,h)}}{Z}$$

and so,

$$Z = \sum_x \sum_h e^{-E(x,h)}$$

3.2 Restricted Boltzmann Machines (RBM)

3.2.1 Metropolis - Hastings algorithm

Metropolis Hastings algorithm is a Markov chain Monte Carlo (MCMC) method for generating random numbers from a probability distribution for which direct sampling is difficult.

An MH step of invariant distribution p(x) and proposal distribution $q(x^*|x)$ involves sampling a value x^* given the current value x according to $q(x^*|x)$. Generated value has a Acceptance probability of $A(x, x^*)$ given below, otherwise remains at x.

$$A(x, x^*) = min\{1, \frac{p(x^*)q(x|x^*)}{p(x)q(x^*|x)}\}$$

3.2.2 Gibbs sampling

Gibbs sampling is a Markov chain Monte Carlo (MCMC) method for generating random numbers which are approximated from a specified multivariate probability distribution, when direct sampling is difficult. It is a randomized algorithm (i.e. an algorithm that makes use of random numbers), and is an alternative to deterministic algorithms for statistical inference such as the expectation-maximization algorithm (EM).

The main idea behind Gibbs sampling is that given a multivariate distribution it is simpler to sample from a conditional distribution than to marginalize by integrating over a joint distribution. Suppose we want to obtain N samples from $\mathbf{x} = (x_1, \ldots, x_n)$ from a joint distribution $p(x_1, \ldots, x_n)$. Denote the ith sample by $\mathbf{x}^{(i)} = (x_1^{(i)}, \ldots, x_n^{(i)})$. We proceed as

1. Initialize $\mathbf{x}^{(i)}$ with some random values.

2. for i = 0 to N-1

   Sample $x_1 \sim p(x_1 | x_2^{(i)}, x_3^{(i)}, \ldots, x_n^{(i)})$
   Sample $x_2 \sim p(x_2 | x_1^{(i+1)}, x_3^{(i)}, \ldots, x_n^{(i)})$

   .

   .

   .

   Sample $x_n \sim p(x_n | x_1^{(i+1)}, x_2^{(i+1)}, \ldots, x_{n-1}^{(i+1)})$

3.2.3 RBM

The RBM is EBM with Energy function $E(x, h)$ as:

$$E(x,h) = -h'Wx - b'x - c'h \tag{3.2}$$

$$p(x,h) = \frac{e^{-E(x,h)}}{Z} \tag{3.3}$$

where W corresponds to the weights connecting hidden and visible units and b and c are the offsets of the visible and hidden layers respectively. The distribution P becomes the Boltzmann distribution. Also, visible and hidden units are conditionally independent given one another. So, we can write:

$$p(h|x) = \prod_i p(h_i|x)$$

$$p(x|h) = \prod_j p(x_j|h)$$

For RBMs with binary units, we obtain the following probabilistic result:

$$P(h_i = 1|x) = \sigma(c_i + W_i x)$$
$$P(x_j = 1|h) = \sigma(b_j + W'_j h)$$

Proof:

From conditional distribution,

$$\begin{aligned} P(h|x) &= \frac{P(x,h)}{P(x)} \\ &= \frac{e^{-(h'Wx + c'h + b'x)}}{\sum_h e^{-(h'Wx + c'h + b'x)}} \\ &= \frac{\prod_{j=1}^{H} e^{-(h_j W_j x + h_j c_j)}}{\prod_{j=1}^{H} \sum_{h_j} e^{-(h_j W_j x + h_j c_j)}} \\ &= \frac{\prod_{j=1}^{H} e^{-(h_j W_j x + h_j c_j)}}{\prod_{j=1}^{H} (1 + e^{-(h_j W_j x + h_j c_j)})} \end{aligned}$$

Therefore, $p(h^{(i)} = 1|x) = \frac{(W_i x + c_i)}{1 + e^{(W_i x + c_i)}}$

and, we have $P(h_i = 1|x) = \sigma(c_i + W_i x)$

and, similarly, $P(x_j = 1|h) = \sigma(b_j + W'_j h)$

The negative log-likelihood of Boltzmann distribution with parameters θ is:

$$-\frac{\partial}{\partial\theta} log(P(x)) = E_{h|x}(\frac{\partial}{\partial\theta} E(x,h)) - E_{x,h}(\frac{\partial}{\partial\theta} E(x,h)) \qquad (3.4)$$

Above gradient contains two terms which are referred as **postive** and

negative phase. Positive phase increases the probability of training data and another decreases the probability of generated samples.

Proof:

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

Differentiating,

$$-\frac{\partial}{\partial\theta} logP(x) = \frac{\partial}{\partial\theta}(-logP(\sum_h \frac{e^{-E(x,h)}}{Z})$$

$$= \frac{-1}{\sum_h P(x)}[\frac{1}{Z}\sum_h(-\frac{\partial E(x,h)}{\partial\theta}e^{-E(x,h)}) - \frac{1}{Z^2}(\sum_h e^{-E(x,h)}\frac{\partial Z}{\partial\theta})]$$

And,

$$\frac{\partial Z}{\partial\theta} = \sum_x\sum_h e^{-E(x,h)}\frac{\partial E(x,h)}{\partial\theta}$$

So,

$$-\frac{\partial}{\partial\theta} logP(x) = (\sum_h \frac{\partial E(x,y)}{\partial\theta}\frac{P(x,h)}{P(x)}) - \sum_x\sum_h \frac{\partial E(x,h)}{\partial\theta}P(x,h)$$

Hence the result follows :

$$-\frac{\partial}{\partial\theta} log(P(x)) = E_{h|x}(\frac{\partial}{\partial\theta}E(x,h)) - E_{x,h}(\frac{\partial}{\partial\theta}E(x,h))$$

where:

$E_{h|X}$ is conditional expectation and $E_{x,h}$ is expectation under joint distribution of x and h.

Now, to find these expectations, we estimate them using some MonteCarlo Techniques. The idea is to form a stationary Markov Chain which converges in distribution to the Boltzmann distribution. Further by Gibbs sampling, we generate both x and h.

So, expectations are estimated as under:

$$E_{h|x}(\frac{\partial}{\partial\theta}E(x,h)) = \frac{\partial}{\partial\theta}E(x,h)$$
$$E_{x,h}(\frac{\partial}{\partial\theta}E(x,h)|x) = \frac{\partial}{\partial\theta}E(\tilde{x},\tilde{h})$$

From Gibbs sampling we have:

$$h^{(n+1)} \sim \sigma(W'x^{(n)} + c)$$
$$x^{(n+1)} \sim \sigma(Wh^{(n+1)} + b)$$

Lastly, we need gradient of negative log-likelihood with respect to the weights W and constants b and c (bias constants). From results we obtain the gradient as under:

$$-\frac{\partial}{\partial W_{jk}}logP(x) = h^{(t)}(x^{(t)})' - \tilde{h}\tilde{x'}$$
$$-\frac{\partial}{\partial b_j}logP(x) = x^{(t)} - \tilde{x}$$
$$-\frac{\partial}{\partial c_k}logP(x) = h^{(t)} - \tilde{h}$$

Bibliography

[1] Yoshua Bengio. Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1):1–127, January 2009.

[2] Yoshua Bengio. Deep learning of representations: Looking forward. *CoRR*, abs/1305.0445, 2013.

[3] Genevieve B. Orr Gregoire Montavon and Klaus-Robert Muller (Eds.). *Neural Networks: Tricks of the Trade.* Second Edition. Springer, 2012.