10-708: Probabilistic Graphical Models, Spring 2015

25 : Deep Learning and Graphical Models

Lecturer: Eric P. Xing *Scribes: Harry Gifford, Pradeep Karuturi*

"Deep learning is just a buzzword for neural nets, and neural nets are just a stack of matrix-vector multiplications, interleaved with some non-linearities. No magic there."

Ronan Collobert

1 Introduction

There has been a recent surge of research in a new class of machine learning techniques called Deep Learning. These techniques are able to achieve state-of-the-art results in prediction and sequence labeling tasks across many fields such as natural language processing, computer vision and tradition classification tasks. Deep Learning is inspired by how the human brain works. The way human visual cortex works is that, when light hits the retina, it goes throgh several regions of the brain and each of these different regions is responsible for extracting different types of information. People started with multi-layer neural networks as a way of modelling this multilayer representation. So, Deep learning models are just network models with multiple hidden layers between the input and the output layers. Though multi-layer feed forward networks were the most popular deep learning models, there are a wide range of deep learning models such as Deep Boltzmann machines, Long Short Term Memory networks(LSTMs) etc.

More formally, Deep learning refers to a class of machine learning techniques, where many layers of information processing stages in hierarchical architectures are exploited for pattern classification and for feature representation[1]. Deep Boltzman Machines(DBN), Deep Neural Networks(DNN) and Deep auto-encoder - all fall under this category of learning techniques. They differ from traditional machine learning approaches in the sense that there is a lot of feature engineering involved in the traditional techniques, deep learning techniques learn their own relevant features. Apart from classification and regression tasks, they can also be used in the case of sequential data. Long Short Term Memory Networks(LSTM) and the general Recurrent Neural Networks(RNNs) are used to model and label sequential data.

Throughout this document we will keep a running example, namely that of classification. The goal of classification is given some data $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$ where $\mathbf{x}^{(i)} \in \mathbb{R}^n$ and $y^{(i)} \in \{-1, 1\}$ we want to learn parameters $\mathbf{w} \in \mathbb{R}^n$ such that $f(\mathbf{w}, \mathbf{x}) : \mathbb{R}^n \times \mathbb{R}^n \to \{-1, 1\}$ predicts y well. Deep learning can of course be used for much more general tasks, but it is good to keep a concrete example in mind.

2 Classic Neural Networks

We start by looking at some of the earliest research on neural networks, since the early research is still very important and acts as a starting point for understanding the state of the art in deep learning.

2.1 Perceptron

The perceptron was one of the earliest breakthroughs in neural inspired learning. The original derivation of the perceptron algorithm is somewhat complicated, but it can be thought of as running stochastic gradient descent on the following cost function, traditionally with no regularization (i.e. $\lambda = 0$).

$$\underset{\mathbf{w}}{\text{minimize}} \quad \frac{\lambda}{2}\|\mathbf{w}\|_2^2 + \sum_{i=1}^{m} \max(0, -y^{(i)}\mathbf{w}^\top \mathbf{x}^{(i)}) \tag{1}$$

Initially there was wide excitement about the perceptron, and this quickly descended into unreasonable excitement about the perceptron. In fact, one group of researchers claimed to train a perceptron to see camouflaged tanks in photos with high accuracy. This might seem surprising given that we still struggle to distinguish between objects in 2015. Indeed it turned out that the perceptron had learned that the pictures with tanks were all developed slightly differently from the pictures without tanks, and so the pictures with tanks were all darker. Therefore the perceptron had simply learned to classify between dark and light images.

After these initial wild claims of the magic, the perceptron problems began to emerge. In particular, Marvin Minsky published the book *Perceptrons* which showed that the single layer perceptron was not rich enough to learn a wide variety of functions. Specifically, the standard perceptron algorithm can only learn a linear decision boundary. Of course we could apply kernels or add extra features, but this style of learning was not well known in the 1960's.

Researchers quickly latched on to the idea of stacking perceptrons in order to learn more complex functions. This approach was discussed in the original work on the perceptron, although the weights of intermediate layers could not be learned and had to be set by hand, limiting their usefulness.

2.2 Logistic Regression

Long before the perceptron was invented a statistician named David Cox developed the Logistic Regression classifier. Logistic Regression can be derived by modeling $p(y \mid \mathbf{x}; \mathbf{w}) = \frac{1}{1+\exp(-y\,\mathbf{w}^\top \mathbf{x})}$. If we evaluate the MAP estimate assuming a Gaussian prior on the $\mathbf{w}$, then we get the following cost function.

$$\underset{\mathbf{w}}{\text{minimize}} \quad \frac{\lambda}{2}\|\mathbf{w}\|_2^2 + \sum_{i=1}^{m} \ln(1 + \exp(-y^{(i)}\mathbf{w}^\top \mathbf{x}^{(i)})) \tag{2}$$

Notice that this cost function is very similar to that of the perceptron. In fact, we can think of it as a smoothed version of the perceptron, where the gradient is defined everywhere. This allows us to use gradient based methods worry free.

It was the key idea that using a smoothed perceptron cost function, such as that given by Logistic Regression, would allow us to stack multiple layers together to create a more powerful Multilayer Perceptron[1]. This was the approach taken by Geoff Hinton in his (in)famous Back-propagation paper.

[1] Unfortunately, the name 'Multilayer Perceptron' is a poor one, since one of the key ideas behind the MLP is to use logistic regression units instead of hard thresholding perceptrons. However, the name has stuck, so we will continue to use it.

2.3 Multilayer Perceptrons

Even with the earliest work on the Perceptron done, Rosenblatt was aware of the limitations of what could be learned by the perceptron. He was also aware that stacking perceptrons could overcome some of the problems. However, it would take 20 years for researchers to figure out how to train a multi-layer perceptron.

The key insight is to use the smoothed logistic regression units, which allow us to get well defined gradients which we can use to recursively update intermediate layers. This recursive gradient descent is known as back-propagation.

In Multilayer perceptrons there are three main components we can control.

1. **Output loss function:** We have to compare $a^{(l)}$, the neural network's predicted label with the true label y. There are many choices, but the most common are squared error $\|a^{(l)} - y\|_2^2$, logistic loss $\ln(1 + \exp(-ya^{(l)}))$ and hinge loss $\max(0, 1 - ya^{(l)})$.

2. **Activation function:** The key to representing arbitrary bounded functions is the non-linear activation functions. While most common activation functions are sufficiently rich to express any bounded function, different activation functions can impact the difficulty of training. Common activation functions are the sigmoid given by $f(x) = (1 + \exp(-x))^{-1}$, tanh, softplus given by $f(x) = \ln(1 + \exp(x))$ and recti-linear given by $f(x) = \max(0, x)$. The latter two are the most popular these days due to their improved training properties. Some of these activation units are shown below.

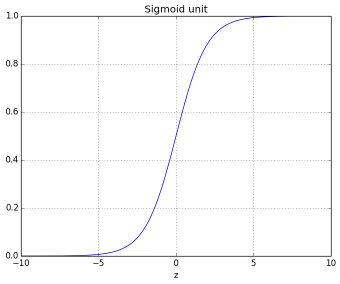

Figure 1: Logistic unit

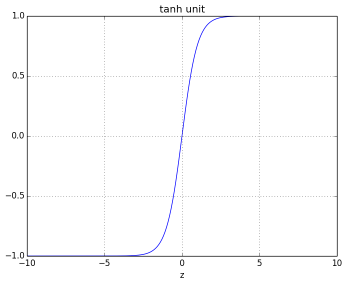

Figure 2: Tanh unit

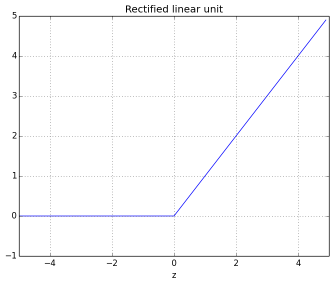

Figure 3: ReLU unit

3. **Structure of the network:** Once we decide on the loss function and activation function, we need to decide on the arichecture of the network, for example: number of layers, number of hidden units etc.

Multilayer perceptrons are also called as feed-forward neural networks. A simple feed-forward neural network with one hidden layer is shown in figure 4. The leftmost layer is the input layer and the rightmost layer is the output layer. This network has parameters $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$, where $W_{ij}^{(l)}$ denotes the weight associated with the connection between unit j in layer l, and unit i in layer $l+1$. b is the bias term. Assume that the activation unit is a *tanh* function which has the range $[-1, 1]$. We can write down the equations for the computation of this neural network as shown below:

$$a^{(2)} = f(W^{(1)}x + b^{(1)}) = f(z^{(2)}) \tag{3}$$

$$h_{W,b}(x) = a^{(3)} = f(W^{(2)}a^{(2)} + b^{(2)}) = f(z^{(3)}) \tag{4}$$

Now the problem is to learn the parameters (W, b) from the given training data, and backpropagation is the standard traning algorithm for learning these parameters in neural networks.

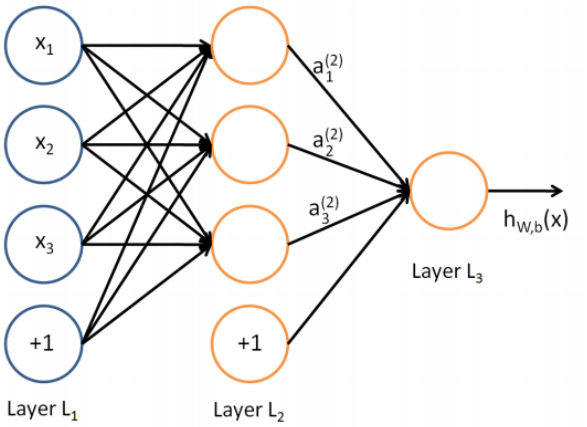

Figure 4: single hidden layer neural network[2]

3 Backpropagation

The main challenge while learning the parameters of a neural network is to learn the weights of the hidden nodes. We can update the weights of the nodes in the final layer using gradient descent techniques. But how do we update the weights of the hidden nodes? Backpropagation is a way to overcome this problem. Given a training example (x, y), we will first run a forward pass to compute all the activations throughout the network, including the output value of the hypothesis $h_{W,b}(x)$. Then, for each node i in layer l, we would like to compute an error term $\delta_i^{(l)}$ that measures how much that node was responsible for any errors in the output[2]. For a node in the output layer, we can measure the difference between the network's activation and the true value and use that to define $\delta_i^{(n_l)}$, where n_l denotes the output layer. Now, this error is propagated backwards into the hidden layers to get $\delta_i^{(l)}$.

Backpropagation is a gradient descent technique where the errors are propagated backwards into the network to update the weights of the hidden nodes. In the case of recurrent neural networks, the errors are propagated backwards in time. This procedure is called backpropagation as we are propagating errors backwards.

Suppose we have a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$ of m training examples and we have a squared error loss as the cost function. Let us define the cost function with respect to a single example to be:

$$J(W, b; x, y) = \frac{1}{2}\|h_{(W,b)}(x) - y\|_2^2 \tag{5}$$

So, the overall cost function for m training example becomes:

$$J(W, b) = \left[\frac{1}{m}\sum_1^m J(W, b; x^{(i)}, y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_i-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{jl}^{(l)})^2 \tag{6}$$

$$= \left[\frac{1}{m}\sum_1^m(\frac{1}{2}\|h_{(W,b)}(x^{(i)}) - y^{(i)}\|_2^2)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_i-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{jl}^{(l)})^2 \tag{7}$$

The first term is the error term and the second term is the regularization term. For the backpropagation, we just do recursive gradient descent over the layers. One iteration of the gradient descent updates the

parameters as follows:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) \tag{8}$$

$$b^{(l)} = b^{(l)} - \alpha \frac{\partial}{\partial b^{(l)}} J(W,b) \tag{9}$$

where α is the learning rate. The partial derivative terms on the right hand side of the equations are very difficult to compute and this is where backpropagation comes into the picture. Backpropagation can be described in four steps[2]:

1. Perform a feedforward pass, computing the activations for all the layers up to the output layer L_{n_l}.
2. For each output unit i in output layer, set

$$\delta_i^{n_l} = \frac{\partial}{\partial z_i^{(l)}} \frac{1}{2} ||h_{W,b}(x) - b||^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)}) \tag{10}$$

3. For $l = n-1, n-2, ..., 2$, for each node i in layer l, set

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)}) \tag{11}$$

4. Compute the desired partial derivatives:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = a_j^{(l)} \delta_i^{(l+1)} \tag{12}$$

$$\frac{\partial}{\partial b^{(l)}} J(W,b) = \delta_i^{(l+1)} \tag{13}$$

Before starting the backpropagation algorithm it is essential to initializae the weights W randomly. Otherwise, all the hidden layer units will end up learning the same function of the input. Even though we showed backpropagation for feedforwarrd neural networks, it can be used for other networks such as stacked autoencoders.

3.1 Practical issues

As mentioned in the previous section, different non-linear activation functions can be used as the basic units of Neural Networks. Some such functions are Logistic(sigmoid), tanh and Rectified Linear Unit(ReLU) functions. Sigmoid function takes in a real value and squashes it into a range between 0 and 1 as shown in figure 1. If we look at the gradient of the sigmoid function in figure 5, we can see that the gradient is almost zero on either tail of the sigmoid function. So, when the value of a sigmoid unit saturates at either tail of 0 or 1, it will kill the gradient and the learning rate in the backpropagation phase becomes very slow leading to the problem of vanishing gradients. This problem becomes more pronounced especially in very deep neural networks, making the training of deep networks very hard. Tanh activation units also suffer from the same problem. So, Rectified Linear Units(ReLU) have become more popular due to their faster convergence rates compared to sigmoid and tanh units. The gradient function for ReLU can be seen in figure 7.

Another practical issue while training deep neural networks is the problem of overfitting. Dropout is a technique to overcome the overfitting problem by randomly dropping(based on a dropout factor) the output

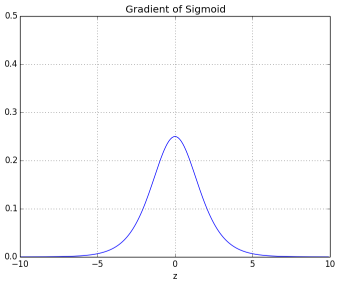

Figure 5: Logistic gradient

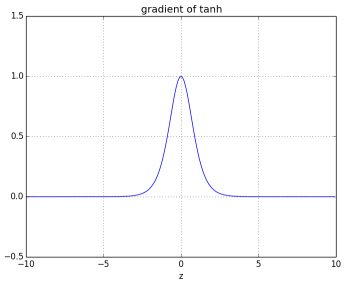

Figure 6: Tanh gradient

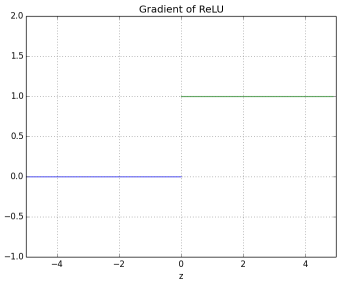

Figure 7: ReLU gradient

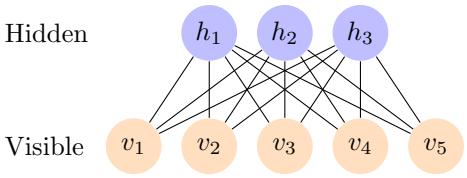

Figure 8: A simple RBM with 5 visible units and 3 latent units.

from some nodes at each layer. Combined with the above techniques and recent improvements in hardware and GPUs, it is relatively easier to train deep neural networks on large amounts of data than it was before. Minibatch Stochastic gradient descient is usually used during the backpropagation to take advantage of the multiple cores and GPUs. Another major problem to the application of DNNs and related deep models is that it currently requires considerable skills and experience to choose sensible values for hyper parameters such as the learning rate schedule, the strength of the regularizer, the number of layers and the number of units per layer, etc.

4 Graphical models

Apart from neural networks, other graphical models are also used in deep learning. For example, Deep Boltzmann Machines(DBN) are based on the basic Restricted Boltzmann machines(RBMs) and it would be quite helpful to understand RBMs to understand Deep Boltzmann Machines.

4.1 Restricted Boltzmann Machines

The Restricted Boltzmann Machine (RBM) is an undirected graphical model introduced in [3]. It represents one possible probabilistic framework to represent a single layer neural network. The idea is that we represent a probability distribution over the input features, using a set of latent (hidden) variables to represent the dependencies between features.

We show an RBM in Figure 8. Notice that each hidden unit is connected with each visible unit and vice versa. We therefore have a bipartite graph. What makes an RBM especially useful is that the hidden units are conditionally independent of one another given the visible nodes and vice versa. This can easily be seen by u-separation. This makes inference especially easy.

4.1.1 Structure

Recall that in an undirected graphical model, we must define a set of potential functions over the cliques of the graph. For an RBM there are only pairwise cliques which simplifies matters. For simplicity let's consider the case where v_i and h_i are Bernoulli. Then

$$\ln p(v) = -F(v) - \ln Z \tag{14}$$

where

$$F(v) = -\sum_i \ln \sum_{h_i} \exp(h_i W_i v) \tag{15}$$

and

$$Z = \sum_v e^{-F(v)}. \tag{16}$$

4.1.2 Training

The benefit of the RBM being a probabilistic model is that we have an interpretable cost function to work with, unlike in the standard neural network. In particular, since we learn a model of $p(x)$ we can train the network by maximizing the log-likelihood:

$$L(\theta \mid X) = \sum_{i=1}^{m} \ln p(x_i) \tag{17}$$

$$= -\sum_{i=1}^{m} F(x) + \ln Z \tag{18}$$

This gives us a very simple cost function, but unfortunately we cannot directly compute the gradient of the RBM since it is intractable for any even mildly large graph. Therefore we must approximate the gradient with samples from $p(v \mid h)$ and $p(h \mid v)$.

4.1.3 Sampling

The most common approach to sampling in an RBM is to use Block Gibbs Sampling to generate samples of $p(v \mid h)$ and $p(h \mid v)$, where we first sample the visible units given the hiddens and then sample the hidden units given the visibles. If we do this we get the following Gibbs steps:

$$p(h = 1 \mid v) = \frac{1}{1 + \exp(-Wv)} \tag{19}$$

$$p(v = 1 \mid h) = \frac{1}{1 + \exp(-Wh)} \tag{20}$$

Notice that we have the same sigmoid function that we had in the standard Neural Network.

4.2 Greedy Pretraining

While RBMs are interesting probabilistic models, in this simple format they are not particularly useful due to their lack of depth. However if we notice that a standard Neural Network looks very similar to a stack of RBMs. Therefore, we can train an RBM on the input to one layer of a neural network and use the trained W matrix from the RBM to initialize the training of a deeper Neural Network. The idea is that the RBM will learn a useful representation of the data in an unsupervised manner, which will also be a useful representation for supervised learning.

5 Deep learning architectures

In the previous sections, we have seen some basic units of the deep learning models. But most of the deep learning techniques are about coming up with new architectures and pipelines. [1] classifies deep learning architectures into three broad classes:
1) Generative deep architectures, which are intended to characterize the high -order correlation properties of the observed or visible data for pattern analysis or synthesis purposes, and/or characterize the joint statistical distributions of the visible data and their associated classes. In the latter case, the use of Bayes rule can turn this type of architecture into a discriminative one. These models are generally used for unsupervised feature learning. Deep Boltzmann Machines(DBMs) and Sum-product Networks(SPN) are good examples of this type of arthictecture.

2) Discriminative deep architectures, which are intended to directly provide discriminative power for pattern classification, often by characterizing the posterior distributions of classes conditioned on the visible data. Convolutional Neural Networks(CNNs) and deep strcutured CRFs are examples of this type of architecture. A sample CNN is shown in figure 9

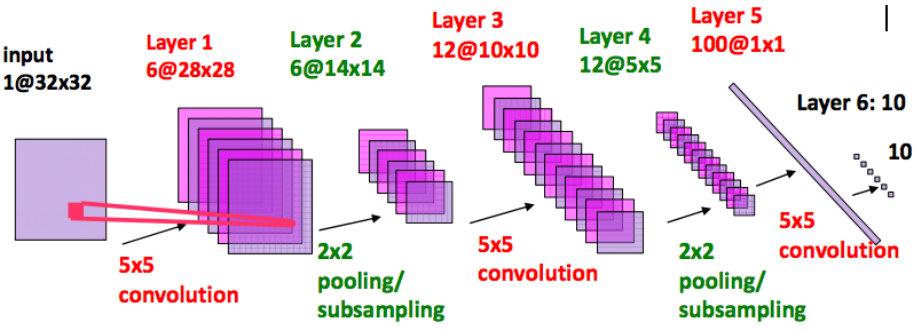

Figure 9: Convolutional Neural Network(CNN)

3) Hybrid deep architectures, where the goal is discrimination but is assisted with the outcomes of generative architectures via better optimization or/and regularization, or when discriminative criteria are used to learn the parameters in any of the deep generative models. For example, in recent times, it is very common to use "pre-trained" Convolutional Neural Networks, whose parameters are initialized by unupservised techniques as the first step of a much bigger problem. DNN-CRF is one example for this type of architecture.

To conclude, deep learning models have been shown to be very effective in a wide variety of machine learning tasks. One of the primary reasons for that is because deep learning techniques learn the essential features from the training data itself, rather than using carefully handcrafted features. This can also be used to get a more compact representation of the data as is done using autoencoders. Deep Neural Networks offer a more compact representation of the models compared to other non-linear models such as non-linear logistic regression which can be exponential in the number of input dimensions.

References

[1] Li Deng *A Tutorial Survey of Architectures, Algorithms, and Applications for Deep Learning*, APSIPA Transactions on Signal and Information Processing, 2013.

[2] Andrew Ng *Notes on Sparse Autoencoders* `https://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf`

[3] Smolensky, P. *Information Processing in Dynamical Systems: Foundations of Harmony Theory, 1986*