10-708: Probabilistic Graphical Models, Spring 2020

11: Foundations of Deep Learning

Lecturer: Eric P. Xing *Scribe: Michael Kronovet, Shreyas Chaudhari, Ahmed Shah*

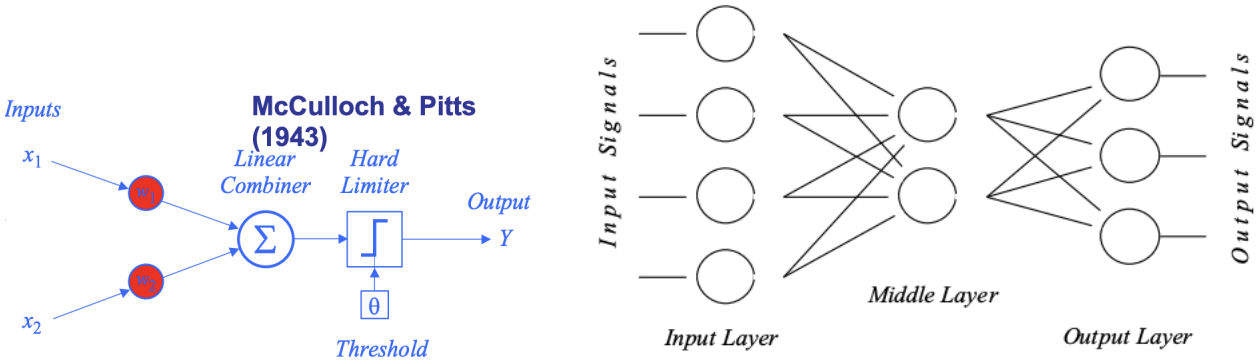
1 An overview of DL components

Many people used to view machine learning as more of a black box. You throw data in a sort of "learning machine", where you tweak some tuning parameters, and get an output. Now we are much more familiar with the underlying math of machine learning models and the underlying mechanisms of how they work.

1.1 Historical Remarks: Early Days of Neural Networks

In the human brain we have roughly 10 billion neurons. Important components of the neuron include dendrites, axons, and synapses. Dendrites are extensions from the neuron cell body that take information to the cell body. They receive stimulation in order for the cell to become active. Axons are the extension from the neuron cell body that takes information away from the cell body. The axon conducts electrical impulses known as action potentials away from the nerve cell body. The function of the axon is to transmit information. Synapses are junctions between nerve cells, and they control the aggregation of signals.

Neural networks were originally an attempt to replicate neurons. In 1943, McCulloch and Pitts published a paper that detailed an artificial neuron (really a perceptron) that worked by mimicking the functionality of a biological neuron. In their model, the output is a hard threshold of a linear combination of inputs. In order to replicate an entire biological neural network, people adapted the McCulloch-Pitts model to have multiple linear combinations of inputs.



1.1.1 Perception

$$\hat{f}(x_i; \vec{w}) = \frac{1}{1 + \exp(-\sum_{i=0}^{n} w_i x_i)}$$

We can imagine that every dendrite will have different conductivity properties, hence the different weights for each input. We seek the weights that minimize the squared residual loss. So, $\vec{w} = \arg\min_{\vec{w}} \sum_i \frac{1}{2}(y_i - \hat{f}(x_i; \vec{w}))^2$

$$\begin{aligned}
\frac{\partial E_D[\vec{w}]}{\partial w_j} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_l (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_l 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_l (t_d - o_d)(-\frac{\partial o_d}{\partial w_i}) \\
&= -\sum_l (t_d - o_d)\frac{\partial o_d}{\partial net_i}\frac{\partial net_d}{\partial w_i} \\
&= -\sum_l (t_d - o_d) o_d(1 - o_d) x_d^i
\end{aligned}$$

where x_d = input, t_d = target output, w_i = weight i, net = $\sum_i w_i x_i$, o_d = observed output = $\frac{1}{1+\exp(\text{- net})}$.

The perceptron learning algorithm is essentially gradient descent. We can either batch update the weights (compute gradient over the entire dataset) or incrementally update the weights (compute gradient for random points). It is better to use incremental updates when the dataset is large, because then compute the gradient over the entire dataset is computationally expensive.

Batch update rule: $\vec{w} = \vec{w} - \eta \nabla E_D[\vec{w}]$ since it calculated across the entire dataset D.

Incremental update rule: $\vec{w} = \vec{w} - \eta \nabla E_d[\vec{w}]$ since it is calculated stochastically for different datapoints $d \in D$.

1.2 Reverse-Mode Automatic Differentiation (Backpropagation)

Similar to the perceptron learning rule, we use gradient descent to solve for the optimal weights in a neural network. Gradient descent in a neural network relies on backpropagation, which is essentially just applying the chain rule from calculus and using reverse accumulation.

We can apply the chain rule and sum over all states of the intermediate variable indefinitely until we reach the value in question we wish to take the derivative with respect to (let's say its x in this case.

$$\frac{\partial f_n}{\partial x} = \sum_{i_1 \in \pi(n)} \frac{\partial f_n}{\partial f_{i_1}}\frac{\partial f_{i_1}}{\partial x} = \sum_{i_1 \in \pi(n)} \frac{\partial f_n}{\partial f_{i_1}} \sum_{i_2 \in \pi(i_1)} \frac{\partial f_{i_1}}{\partial f_{i_2}}\frac{\partial f_{i_2}}{\partial x} = \ldots$$

This process is fully automated in modern programs like tensorflow.

1.3 Modern Building Blocks: Units, Layers, Activation Functions, Loss Functions, Etc.

1.3.1 Activation Functions

Activation functions define the output of a node in a neural network given a linear combination of the previous layer's outputs. Common activation functions are the linear activation $f(x) = x$, the ReLu activation $f(x) = \max\{0, x\}$, the sigmoid activation $f(x) = \frac{1}{1+\exp(-x)}$, the tanh activation $\frac{e^x - e^{-x}}{e^x + e^{-x}}$, etc.

1.3.2 Structure of Networks

The structure of a neural network defines the underlying computation of what that network is doing. There are many different network structures that have been adapted and specialized for specific tasks. Convolutional neural networks (CNNs) have been used for image processing, recurrent neural networks (RNNs) have often been used to work with time series data, and there are many other formulations of neural networks that are optimized for certain use cases.

1.3.3 Loss Functions

Loss functions are imperative for the training of neural networks because they tell your weights how to update during backpropagation. Different loss functions are preferable in different problem settings. For example, if we were interested in training a neural network to predict a class label for a set of inputs, then cross-entropy could be a good loss function for that network. When modeling a classification problem where we are interested in mapping input variables to a class label, we can model the problem as predicting the probability of an example belonging to each class, which is why cross entropy, which measures the error between two probability distributions, could be a good choice. On the other hand, using a loss function like mean squared error that would take the squared difference between output classes may lead the network to perform more poorly. Suppose you choose a loss function for your neural network that optimizes for the squared difference between your predicted label and the actual label. Then, if you incorrectly classify something as belonging to class 5 instead of class 1, that would have a much more impactful error on your model than if you were to predict class 2 instead of class 1. Depending on how you would want to weight these errors, you would want to tweak your loss function.

Neural networks can use any arbitrary combinations of activation functions, layer structures, and loss functions. Interestingly, it seems that given enough data deeper network structures keep improving their predictive abilities. We believe this is the case because deeper networks with more training allow the networks to learn increasingly more abstract representations of the data. For example, when a neural network is trained on identifying faces, the weights on the deeper network layers when observed start to take on facial features. In fact, the deeper into the network you go the more complex and detailed the faces and patterns in the weights become. Since neural networks are capable of "disentangling" the data to learn complex representations of features from the data, they are often referred to as performing representation learning.

2 Similarities and differences between GMs and NNs

Both graphical models and GMs have similarities in their empirical goals, structure (graphical) and objective (aggregated from local functions). But the similarities end there and seemingly similar tasks of inference and learnign have different meanings under the two lenses. These differences have been tabulated below:

	Graphical Models	Deep Neural Networks
Representation	Encode meaningful knowledge and the associated uncertainty in a graphical form	Representations facilitate computation and performance on the end-metric and intermediate representations are not guaranteed to be meaningful
Learning and Inference	Based on a rich toolbox of well-studied and structure-dependent techniques like EM, message passing, VI, MCMC, etc.	Learning is predominantly based on the gradient descent method (backpropagation); Inference is often trivial and done via a "forward pass"
Structure	Graphs represent models	Graphs represent computation

As the above table summarizes, though both GMs and NNs have a graphical structure the graphs have different utilities. Utility of the graph for **graphical models**:

- Synthesizing a global loss function from local structure
- Designing sound and efficient inference algorithms
- Inspire approximation and penalization
- Monitoring theoretical and empirical behavior and accuracy of inference

Utility of the graph for **deep neural networks**:

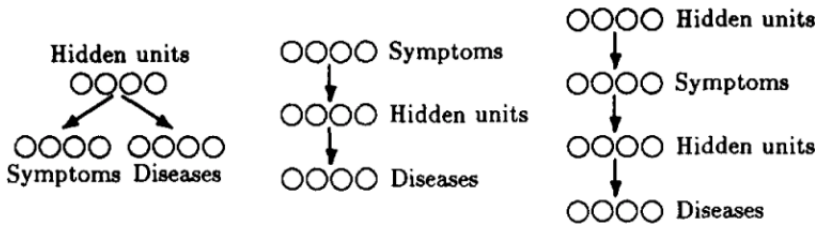
- Conceptually synthesize complex decision hypothesis
- Organizing computational operations
- Designing processing steps and computing modules
- Inference algorithms in DL have no obvious utility

This comparison can be summed up by these two statements - Graphical models are representations of probability distributions. Neural networks are function approximators (with no necessary probabilistic meaning).

The following section considers some of the neural nets that are in fact proper graphical models.

3 Neural Networks That Are Graphical Models

3.1 Sigmoid Belief Networks



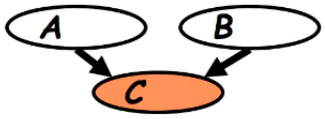
Sigmoid belief networks have been around for around 30 years and in the simplest form, they have one hidden layer with conditional probabilities represented by sigmoid functions. This does not allow for complex combination of features/nodes. Adding a or multiple hidden layers can allow for such combinations (the figure).

Sigmoid belief nets are simply Bayesian networks over binary variables with conditional probabilities represented by sigmoid functions:

$$P\left(x_i|\pi\left(x_i\right)\right) = \sigma\left(x_i \sum_{x_j \in \pi(x_i)} w_{ij} x_j\right)$$

3.1.1 Explaining away effect

Bayesian networks exhibit a phenomenon called "explain away effect". This is a consequence of conditional dependence in directed graphical models for V-structures. When conditioned on visible variables, the hidden variables are dependent on each other.



This coupling or correlation creates a computational difficulty for sigmoid belief networks.

3.1.2 Learning and Inference

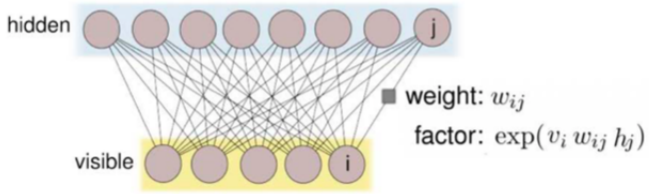
The learning in sigmoid belief networks has the update rule given in the equations that follow (proposed by Neal, 1992).

$$\begin{aligned}
\frac{\partial L}{\partial w_{ij}} &= \sum_{\tilde{v}\in\mathcal{T}} \frac{1}{P(\tilde{V}=\tilde{v})} \frac{\partial P(\tilde{V}=\tilde{v})}{\partial w_{ij}} \\
&= \sum_{\tilde{v}\in\mathcal{T}} \frac{1}{P(\tilde{V}=\tilde{v})} \sum_{\tilde{h}} \frac{\partial P(\tilde{S}=\langle\tilde{h},\tilde{v}\rangle)}{\partial w_{ij}} \\
&= \sum_{\tilde{v}\in\mathcal{T}} \sum_{\tilde{h}} P(\tilde{S}=\langle\tilde{h},\tilde{v}\rangle|\tilde{V}=\tilde{v}) \cdot \frac{1}{P(\tilde{S}=(\tilde{h},\tilde{v}))} \frac{\partial P(\tilde{S}=\langle\tilde{h},\tilde{v}\rangle)}{\partial w_{ij}} \\
&= \sum_{\tilde{v}\in\mathcal{T}} \sum_{\tilde{s}} P(\tilde{S}=\tilde{s}|\tilde{V}=\tilde{v}) \frac{1}{P(\tilde{S}=\tilde{s})} \frac{\partial P(\tilde{S}=\tilde{s})}{\partial w_{ij}} \\
&= \sum_{\tilde{v}\in\mathcal{T}} \sum_{\tilde{s}} P(\tilde{S}=\tilde{s}|\tilde{V}=\tilde{v}) \frac{1}{\sigma\left(s_i^* \sum_{k<i} s_k w_{ik}\right)} \frac{\partial \sigma\left(s_i^* \sum_{k<i} s_k w_{ik}\right)}{\partial w_{ij}} \\
&= \sum_{\tilde{v}\in\mathcal{T}} \sum_{\tilde{s}} P(\tilde{S}=\tilde{s}|\tilde{V}=\tilde{v}) s_i^* s_j \sigma\left(-s_i^* \sum_{k<i} s_k w_{ik}\right)
\end{aligned}$$

where the last expression is approximated with Gibbs sampling. Due to the *explaining away effect*, this sampling will be hard. This leads to slow convergence.

3.2 Restricted Boltzmann Machines

RBM is a Markov Random Field represented with a bi-partite graph. All nodes in one layer of the graph are connected to all nodes in the other. There are no intra-layer connections, which induced the conditional independence properties for undirected graphical models.



Usually the visible and hidden variables (v, h) are binary variables, though it can be generalised to Gaussian Bernoulli RBMs. The joint distribution is represented by:

$$P(v,h) = \frac{1}{Z} \exp\left\{ \sum_{i,j} w_{ij} v_i h_i + \sum_i b_i v_i + \sum_j c_j h_j \right\}$$

3.2.1 Learning in RBMs

The log-likelihood of a single data point where the unobservables marginalized out in the last term is:

$$\log L(v) = \log \sum_h \exp\left\{ \sum_{i,j} w_{ij} v_i h_i + \sum_i b_i v_i + \sum_j c_j h_j - \log(Z) \right\}$$

Gradient of the log-likelihood w.r.t. the model parameters, written as expectations is:

$$\frac{\partial}{\partial w_{ij}} \log L(v) = \underbrace{E_{P(h|v)}\left[\frac{\partial}{\partial w_{ij}} P(v,h) \right]}_{\text{over the posterior}} - \underbrace{E_{P(v,h)}\left[\frac{\partial}{\partial w_{ij}} P(v,h) \right]}_{\text{over the joint}}$$

Both expectations are approximated via sampling. This leads to two 'phases':

- Sampling from the posterior is exact (RBM factorizes over h given v): called the clamped / wake / positive phase as the network is "awake" since it conditions on the visible variables
- Sampling from the joint is done via MCMC (e.g., Gibbs sampling): called the unclamped / sleep / free / negative phase as the network is "asleep" since it samples the visible variables from the joint. In some sense it, is "dreaming" the visible inputs

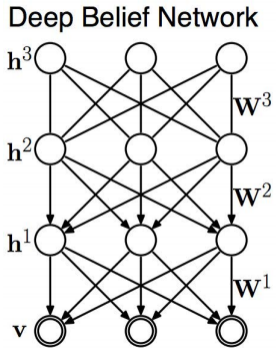
Learning is done by optimizing the log-likelihood of the model for a given data via gradient descent. Estimation in the sleep phase heavily relies on the mixing properties of the Markov chain which causes slow convergence and requires extra computation.
The other way round applies to directed latent variable models, as the following section describes.

3.2.2 RBMs are Infinite Belief Networks

Recall that the gradient of the training objective of RBMs, i.e. the log likelihood of the visible units, contains an expectation over the joint probability, $p(v, h)$. To compute this expectation we sample the joint distribution using block Gibbs sampling, which involves alternatively clamping v and sampling h, and clamping h and sampling v until the samples converge to a stationary distribution. Observe that this process is exactly equivalent to top-down propagation in an *infinitely* deep sigmoid network in which the weights are shared across all the layers. Further observe that a model that stacks a RBM on top of a sigmoid which also shares the same weights is strictly equivalent to the RBM itself. If the weights of the sigmoid network and the RBM are then untied we get a Deep Belief Network, which is explained in detail in the following section.

4 Deep Belief Network

Deep Belief Networks (DBNs) are hybrid graphical models that comprise of a RBM stacked on top of a sigmoid network. Importantly, the weights of the RBM and the sigmoid networks are untied. If they were tied the overall model, as mentioned earlier, would be equivalent to just the RBM.



A DBN represents the following joint probability distribution

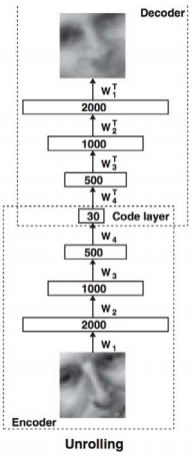
$$P(v, h_1, h_2, ..., h_{k-1}, h_k) = P(v|h_1) \left(\prod_{i=1}^{k-2} P(h_i|h_{i+1}) \right) P(h_{k-1}, h_k)$$

where $P(h_{k-1}, h_k)$ is modeled by an RBM and $P(h_i|h_{i+1})$ are modeled by sigmoid networks. The model is trained by maximizing, $\log P(v)$, the log-likelihood of the given data. Training is done stage-wise, with a greedy layer-wise pretraining step followed by an ad-hoc, possibly task dependent, fine-tuning step.

The layer-wise pretraining proceeds as follows. First, a RBM is trained. The weights of the RBM are then frozen and another RBM is stacked on top of it. The second RBM is trained on the output of the first RBM. Note that the weights of the first and second RBMs are *untied*. Proceeding in this manner we can incrementally add *pretrained* layers to the DBN. Observe that this process is equivalent to untying and tuning the weights of an infinite sigmoid network layer by layer, starting from the bottom and moving up. Common variants of this training process either tune the weights of each RBM until convergence by repeated iterations of the wake and sleep steps, or may perform only one weight update before moving to the next RBM.

Since the pre-training procedure is mostly ad-hoc and does not guarantee convergence to a good probabilistic model, a fine-tuning step is performed to tune the model for some downstream task. The hope is that

the representations learned by the model in pretraining step capture information that can be used in the downstream task. One such task could be to produce a feature representation for images. For this task we can unroll the pretrained DBN to create an *autoencoder* as in the diagram below.



Then we can fine-tune all the weights by backpropagating the reconstruction error between the original image and the reconstructed image. In the diagram above, the code layer provides a representation of the image.

A closely related model to DBNs is the Deep Boltzmann Machine (DBM), which consists of multiple RBMs stacked together. DBMs can be trained like boltzmann machines, using MCMC. Weights from pretrained DBMs can be used to initialize other networks for downstream tasks.

5 Graphical Models vs. Deep Networks

The primary goal of deep networks is to derive a representation of the distribution of random variables (features) using a vast interconnected set of latent variables. Like GMs, adding more layers of hidden/latent variables allows the model to represent more complex distributions, however, unlike GMs, DNs are not concerned with correctly learning and inferring the values of these latent variables. In fact, the representation produced by the hidden variables in a DN is evaluated solely by the model's performance on a downstream task. The hidden variables can take on any value as long as the performance on the downstream task remains optimal.

5.1 Study of Belief Networks as GMs

In [1], the authors compare the performance of mean-field belief propagation and other approximate inference algorithms when applied to deep sigmoid belief networks. The values of the hidden variables were simulated and the error was measured between the approximated values and the true values of the hidden variables. The study found that belief propagation (i.e. back-propagation) performs the worst, while mean-field approximations algorithms perform much better, in terms of, both, time and accuracy.

6 Optimizing Optimization

Recall that a RBM (or any other undirected GM), when unrolled yields infinite instantiations of another GM such that each instantiation shares the same weights. During training, we perform one weight update on each GM instantiation and move to the next until the weights converge. This process is akin to gradient descent, in that we have theoretical guarantees of converging to an optimal point in the limit. In contrast, deep learning models keep updating the weights of each sub-model until convergence before moving on to the next. Therefore, deep learning models are not designed to asymptotically converge and can be seen as optimizing the optimization steps of an undirected GM using backpropagation.

7 Dealing with Structured Prediction in Deep Learning

Structured graphical models, such as CRFs, can be converted to deep learning models by urolling them for a fixed number of steps and optimizing each step using backpropagation. Similarly, message passing based inference algorithms can be trucated and converted into computational graphs.

References

[1] Eric P Xing, Michael I Jordan, and Stuart Russell. A generalized mean field algorithm for variational inference in exponential families. *arXiv preprint arXiv:1212.2512*, 2012.