Deep Learning

February 28, 2017

While deep neural network architectures have been out for many years, they were difficult to train with backpropagation algorithms - the errors would blur after a few layers. Hinton[4] introduced a way to pre-train the network weights based on Restricted Bolzman Machines (RBMs). An RBM converts an input vector of length n into a hidden layer vector of length m. Hinton provides a general introduction and Matlab code[4]. Bengio *et al.* provide detailed algorithms and a proper handling of unit continuous input layers[2]. Bengio also provides an in-depth overview of deep learning[1]. Once it is pre-trained, the nework is unrolled and fine-tuned with standard backpropagation.

1 General definitions

1.1 Energy function

For an RBM, the energy is calculated as:

$$E(\bar{f}, \bar{h}) = -\sum_i b_i f_i - \sum_j c_j h_j - \sum_{ij} W_{ji} f_i h_j \tag{1}$$

Where $\bar{f} = (f_1, \ldots, f_n)$ is the input feature vector, $\bar{b}$ the biases on $\bar{f}$, $\bar{h}$ is the hidden features vector and $\bar{c}$ the biases of $\bar{h}$, and W is the weight matrix where W_{ij} is the weight between f_i and h_j. Hereafter, the vector versions of $\bar{f}$, $\bar{h}$, $\bar{b}$ and $\bar{c}$ will be noted f, h, b and c respectively.

For the whole deep network, given a gaussian input, the following can be computed:

$$E(\bar{f}) = -\sum_i \frac{1}{2} b_i f_i^2 - \sum_{l=1}^{N} \left(\sum_i b_i^l h_i^{l-1} - \sum_j c_j^l h_j^l - \sum_{ij} W_{ji}^l h_i^{l-1} h_j^l \right) \tag{2}$$

1.2 Activation function

The activation function transforms the activations α into activities z. For a given layer l of the network, we can define:

$$z_j^{(l)} = h^{(l)}(\alpha_j^{(l)}) \tag{3}$$

For a sigmoid binary layer, the following function is used:

$$\sigma_b(\alpha) = h_b(\alpha) = \frac{1}{1+e^{-\alpha}} \tag{4}$$

and its derivative is:

$$\sigma_b^{'}(\alpha) = h_b^{'}(\alpha) = \frac{e^{-\alpha}}{(1+e^{-\alpha})^2} \tag{5}$$

And for a continuous sigmoid layer:

$$\sigma_c(\alpha) = h_c(\alpha) = \frac{e^{\alpha}(1-\frac{1}{\alpha})+\frac{1}{\alpha}}{e^{\alpha}-1} \tag{6}$$

$$\sigma_c^{'}(\alpha) = h_c^{'}(\alpha) = \begin{cases} \frac{1}{\alpha^2} - \frac{1}{e^{\alpha}+e^{-\alpha}-2}, & \alpha \neq 0 \\ \frac{1}{12} - \frac{\alpha^2}{240}, & |\alpha| < 10^{-5} \end{cases} \tag{7}$$

This value for the limit of the second derivative was estabished by testing with R on a 64bits MacOS X system.

In our setup we should not need its derivative.
Finally for a gaussian layer:

$$\sigma_g(\alpha) = h_g(\alpha) = \alpha \tag{8}$$

and its derivative:

$$\sigma_g^{'}(\alpha) = h_g^{'}(\alpha) = 1 \tag{9}$$

The activations for layer l are defined as

$$\alpha_j^{(l)} = \sum_{\tilde{i}} W_{j\tilde{i}}^{(l)} z_{\tilde{i}}^{(l-1)} \tag{10}$$

Here $\tilde{i}$ is a dummy index on the i[1]. See also the next section for a slightly different definition of α, where the biases are separated from the weights.

An equivalent form with separate biases b is used during the contrasted divergence:

$$\alpha_j^{(l)} = b_j^{(l)} + W_{j\tilde{i}}^{(l)} z_{\tilde{i}}^{(l-1)} \tag{11}$$

2 Pre-training of the RBM

The RBM can be pre-trained by sampling the hidden node in a positive phase, then resampling a feature vector and a new hidden layer, and using the reconstruction error to compute an update of the weights of the RBM.

[1] i.e it has an effect only when $\tilde{i} = j$ where j is now the indices on the layer $l-1$.

2.1 Computation of hidden layer and feature vector reconstruction

One can derive several probabilities from the energy function (1). The relationship is given by:

$$P(h, f) = \frac{1}{z} e^{-E(f,h)} \tag{12}$$

where Z is a normalization constant.
With a sigmoid activation function, the probability of the hidden layer, given the data f is then expressed as:

$$P(h_j|f) = \frac{e^{\alpha_j h_j}}{1 + e^{\alpha_j}} \tag{13}$$

$$\langle h \rangle_{|f} = P(h_j = 1|f) = \frac{1}{1 + e^{-\alpha_j}} \tag{14}$$

with

$$\alpha_j = c_j + \sum_i W_{ij} f_i \tag{15}$$

For a binary visible layer, the probability is given by:

$$P_b(f_i = 1|h) = \frac{1}{1 + e^{-\beta_i}} \tag{16}$$

where

$$\beta_i = b_i + \sum_j W_{ij} h_j \tag{17}$$

For a unit continuous visible layer, the one must in addition pass an f_i as:

$$P_c(f_i|h) = \frac{\beta_i}{e^{\beta_i} - 1} e^{\beta_i f_i} \tag{18}$$

The expectation value is then:

$$\langle h \rangle_{|f} = \frac{e^{\alpha_i}(1 - \frac{1}{\alpha_i}) + \frac{1}{\alpha_i}}{e^{\alpha_i} - 1} \tag{19}$$

Hinton[4] doesn't describe it in detail, but he trains gaussian layers as gaussian (see Matlab code). Those layers have the following expectation value:
The expectation value is then:

$$\langle h \rangle_{|f} = \alpha \tag{20}$$

2.2 Updating rule

We use gradient descent to update the weight and biases W, b and c. For this, one must compute the following update terms for each data point f^0 iteratively:

$$W' = W + \tanh(\epsilon_W(h^{0^T} f^0 - h^{1^T} f^1)) \tag{21}$$

$$b' = b + \tanh(\epsilon_b(f^0 - f^1)) \tag{22}$$

$$c' = c + \tanh(\epsilon_c(h^0 - h^1)) \tag{23}$$

Where ϵ_x is the learning rate for W, b and c, and W, b and c are initially filled with 0.

h^0 is the hidden vector sampled from f^0 with $P(h = 1|f)$,

f^1 is the expectation value of f given h^0 ($P(f|h^0)$ for binary and unit continuous inputs, α for gaussian inputs), and

h^1 is the expectation value of h given f^1 ($P(h = 1|f)$ for binary and unit continuous outputs, β for gaussian outputs).

Bengio [2] uses $P(h = 1|f^1)$ directly (he calls that Q), instead of h^1.

2.3 Sampling

To resample a hidden unit from a unit continuous feature vector f:

$$h_j = y_j < P(h_j = 1|f) \tag{24}$$

where y_j is a random number drawn from a uniform distribution between 0 and 1.

To resample a binary feature vector from the hidden unit h:

$$f_i = y_i < P_b(f_i = 1|h) \tag{25}$$

To resample a unit continuous feature vector:

$$f_i \sim P_c(f_i|h) \tag{26}$$

so:

$$f_i = \begin{cases} \frac{1}{\beta_i} \log(y_i(e^{\beta_i} - 1) + 1) & \beta_i \neq 0 \\ y_i & |\beta_i| < 10^{-6} \end{cases} \tag{27}$$

where y_i is a random number drawn from a uniform distribution between 0 and 1, and $\beta_i \simeq 0$ when $|\beta_i| < 10^6$.

This value was estabished by testing with R on a 64bits MacOS X system.

Finally to resample a gaussian feature vector we use:

$$f_i = \beta_i + z_i \tag{28}$$

with z_i randomly drawn from a standard normal distribution.

2.4 Penalization

Penalization or weight-decay prevents the weights W, b and c from becoming too large over the training. Hinton [4] penalizes only the weights W, however in our experience we still observed some drift of b and c, so they are penalized as well.

2.4.1 L1

With L1 penalization a few weights will be allowed to become quite large, but most of them will be kept very small. This favors sparsity for the network [3, pp. 145, 146]. L1 penalizes $|W|$, $|b|$ and $|c|$.

The updating rules in equations (21, 22 and 23) are modified as follow:

$$W' = W + \tanh(\epsilon_W(h^{0^T} f^0 - h^{1^T} f^1) - \lambda_w sign(W)) \tag{29}$$

$$b' = b + \tanh(\epsilon_b(f^0 - f^1) - \lambda_b sign(b)) \tag{30}$$

$$c' = c + \tanh(\epsilon_c(h^0 - h^1) - \lambda_c sign(c)) \tag{31}$$

where λ is the penalty for large weights W, b and c.

2.4.2 L2

L2 penalizes W^2, b^2 and c^2. Weights will not be allowed to become large, but they will not be penalized if they are only slightly different from 0.

The updating rules in equations (21, 22 and 23) are modified as follow:

$$W' = W + \tanh(\epsilon_W(h^{0^T} f^0 - h^{1^T} f^1) - \lambda_w W) \tag{32}$$

$$b' = b + \tanh(\epsilon_b(f^0 - f^1) - \lambda_b b) \tag{33}$$

$$c' = c + \tanh(\epsilon_c(h^0 - h^1) - \lambda_c c) \tag{34}$$

2.5 Momentum

We decided not to use momentum. This is basically useless with weight-decay because the weights are already constrained.

2.6 Convergence and stopping

We still have to figure out when to stop the pre-training.

- Early stopping (i.e. after going through all or a given number of the cases in the training sample)

Hopefully we can say we only need to see n cases before we have a reasonable fit - anyway we'll backpropagate in the end.

- Error convergence (slow)
- $log(P_m(X_{data}))$
- Parallel tempering

3 From Restricted Bolzman Machine to Deep Belief Networks

A Deep Belief Networks (DBN) is a sequential stack of l RBMs, where the output of layer $l-1$ becomes the input of layer l and l is relatively large. It is defined by the vector of weights $\bar{W} = \{W^{(1)}, W^{(2)}, ..., W^{(l)}\}$.

The DBM, can be unrolled, i.e. the encoder DBM (that was just pre-trained) is duplicated into a decoder that is reversed (inputs become outputs, W is transposed and b and c biases are exchanged) and added on top of the encoder into an *autoencoder*. As a result the autoencoder has the same output as its input, $2l$ layers.

4 Backpropagation and fine-tuning

Once the DBN is pre-trained and unrolled, it can be fine-tuned with backpropagation. The idea is to compute the error ε of the prediction, and use the gradient of this error $\partial\varepsilon$ to update the weights

4.1 Error computation

In an unrolled autoencoder, the target of the output is the input itself. Thus, if x is the data (noted f in the previous section), y is the output and t is the target we can write $t = f$, and the quadratic error of a unit continuous output becomes:

$$\varepsilon(y,t) = \frac{1}{2}\sum_{\tilde{j}}(y_{\tilde{j}} - t)^2_{\tilde{j}} \tag{35}$$

For a categorical output with k categories:

$$\varepsilon_n = -\sum_{k=1}^{k} t_{nk} ln(y_{nk}) \tag{36}$$

with $\tilde{j}$ as a dummy index. We can express the gradient of the activation j of the last layer L as a function of the error at the last layer L as:

$$\delta_j^{(L)} = \frac{\partial\epsilon}{\partial y_j}\left[h^{(L)}\right]'\left(\alpha_j^{(L)}\right) \tag{37}$$

In the case of a quadratic error function, we then have:

$$\delta_j^{(L)} = (y_j - t_j) \left[h^{(L)}\right]' \left(\alpha_j^{(L)}\right) \tag{38}$$

and for a categorical error function:

$$\delta_j^{(L)} = \frac{-t_j}{y_j} \left[h^{(L)}\right]' \left(\alpha_j^{(L)}\right) \tag{39}$$

From this, we can backpropagate the gradient to the previous layers, for instance $l = L - 1$:

$$\delta_j^{(L-1)} = \left[h^{(L-1)}\right]' \left(\alpha_j^{(L-1)}\right) \sum_i W_{ji}^{(L)} \delta_i^{(L)} \tag{40}$$

or more generally for any layer $l \neq L$:

$$\delta_j^{(l-1)} = \left[h^{(l-1)}\right]' \left(\alpha_j^{(l-1)}\right) \sum_i W_{ji}^{(l)} \delta_i^{(l)} \tag{41}$$

We can now use this gratient of the activations to compute the gradient of the weights:

$$\frac{\partial \epsilon}{\partial W_{ji}^{(l)}} = \delta_j^{(l)} z_i^{(l-1)} \tag{42}$$

The biases are handled similarly, however they don't depend on lower layers and simplify to:

$$\frac{\partial \epsilon}{\partial c_j} = \delta_j^{(l)} \tag{43}$$

From this an optimal update of the weights can be computed with a method such as conjugate gradient optimization. The following map:

$$\frac{\partial \epsilon}{\partial W_{ji}^{(l)}} = \frac{\partial \epsilon}{\partial W_k} \tag{44}$$

References

[1] Yoshua Bengio. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[2] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153–160, 2007.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, August 2006.

[4] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, May 2006.