Neural Networks

Escuela de Ciencias Informáticas 2014

July 11, 2014

Contents

1 Softmax Regression

Source: Ian Goodfellow. *"Pylearn2 Tutorial: Softmax regression"*. `http://nbviewer.ipython.org/github/lisa-lab/pylearn2/blob/master/pylearn2/scripts/tutorials/softmax_regression/softmax_regression.ipynb`

Softmax regression is a type of classification model (so the "regression" in the name is really a misnomer), which means it is a pattern recognition algorithm that maps input patterns to categories. Softmax regression is a generalization of logistic regression to multiple categories.

Let's define some basic terms. First, we'll use the variable x to represent the input to the softmax regression model. We'll use the variable y to represent the output category. Let y be a non-negative integer, such that $0 \leq y < k$, where k is the number of categories x may belong to. We interpret y as being a numeric code identifying a category, e.g., 0 = cat, 1 = dog, 2 = airplane, etc.

The job of the softmax regression classifier is to predict the probability of x belonging to each class. i.e, we want to be able to compute $p(y = i|x)$ for all k possible values of i.

The role of a parametric model like softmax regression is to define a set of parameters and describe how they map to functions f defining $p(y|x)$. In the case of softmax regression, the model assumes that the log probability of $y = i$ is an affine function of the input x, up to some constant $c(x)$. $c(x)$ is defined to be whatever constant is needed to make the distribution add up to 1.

To make this more formal, let $p(y)$ be written as a vector $[p(y = 0), p(y = 1), \ldots, p(y = k-1)]^T$. Assume that x can be represented as a vector of numbers (for example, we will regard each pixel of an grayscale image as being represented by a number in $[0, 1]$, and we will turn the 2D array of the image into a vector). Then the assumption that softmax regression makes is that

$$\log p(y|x) = x^T W + b + c(x)$$

where W is a matrix and b is a vector. Note that $c(x)$ is just a scalar but here I am adding it to a vector. I'm using numpy broadcasting rules in my math here, so this means to add $c(x)$ to every element of the vector.

W and b are the parameters of the model, and determine how inputs are mapped to output categories. We usually call W the "weights" and b the "biases".

By doing some algebra, using the constraint that $p(y)$ must add up to 1, we get

$$p(y|x) = \frac{exp(x^T W + b)}{\sum_i exp(x^T W + b)_i} = \text{softmax}(x^T W + b)$$

where softmax is the *softmax activation function.*

1.1 The basic theory of how softmax regression training works

Of course, the softmax model will only assign x to the right category if its parameters have been adjusted to make them specify the right mapping. To do this we need to train the model.

The basic idea is that we have a collection of training examples, $\mathcal{D}$. Each example is an (x, y) tuple. We will fit the model to the training set, so that when run on the training data, it outputs a good estimate of the probability distribution over y for all of the x's.

One way to fit the model is maximum likelihood estimation. Suppose we draw a category variable $\hat{y}$ from our model's distribution $p(y|x)$ for every training example independently. We want to maximize the probability of all of those labels being correct. To do this, we maximize the function

$$J(\mathcal{D}, W, b) = \prod_{x,y \in \mathcal{D}} p(y|x).$$

That function involves lots of multiplication, of possibly very small numbers (note that the softmax activation function guarantees none of them will ever be exactly zero). Multiplying together many small numbers can result in numerical underflow. In practice, we usually take the logarithm of this function to avoid underflow. Since the logarithm is a monotically increasing function, it doesn't change which parameter value is optimal. It does get rid of the multiplication though:

$$J(\mathcal{D}, W, b) = \sum_{x,y \in \mathcal{D}} \log p(y|x).$$

Many different algorithms can maximize J. For this case, we can use an algorithm called Nonlinear Conjugate Gradient Descent to minimize $-J$. In the case of softmax regression, maximizing J is a convex optimization problem so any optimization algorithm should find the same solution.

One problem with maximium likelihood estimation is that it can suffer from a problem called *overfitting.* The basic intuition is that the model can memorize patterns in the training set that are specific to the training examples, i.e. patterns that are spurious and not indicative of the correct way to categorize new, previously unseen inputs. One way to prevent this is to use *early stopping.* Most optimization methods are iterative, in that they try out several values of W and b gradually looking for the best one. Early stopping refers to stopping this search before finding the absolute best values on the training set. If we start with W close to the origin, then stopping early means that W will not travel as far from the origin as it would if we ran the optimization procedure to completion. Early stopping corresponds to assuming that the correlations between input features and output categories are not as strong as pure maximum likelihood estimation would determine them to be.

In order to pick the right point in time to stop, we divide the training set into two subsets: one that we will actually train on, and one that we use to see how well the model is generalizing to new data, then “validation set”. The idea is to return the model that does the best at classifying the validation set, rather than the model that assigns the highest probability to the training set.

2 Multilayer Perceptrons

Source: Ian Goodfellow. *“Pylearn2 Tutorial: Multilayer Perceptron”*. `http://nbviewer.ipython.org/github/lisa-lab/pylearn2/blob/master/pylearn2/scripts/tutorials/multilayer_perceptron/multilayer_perceptron.ipynb`

2.1 Review of softmax regression, and how MLPs are similar

In Sec. 1, we saw how softmax regression is a classification model that learns to map an input vector x to a probability distribution $p(y|x)$ where y is a categorical value with k different values. We then described how a dataset $\mathcal{D}$ of (x, y) tuples could be used to train a softmax regression model by maximizing the log likelihood,

$$\sum_{x,y \in \mathcal{D}} \log p(y|x).$$

A multilayer perceptron (MLP or Artificial Neural Network - ANN) is a very general machine learning model. In many cases, we can think of it as mapping x to $p(y|x)$, and train it by maximizing the log likelihood. We’ll start with that basic perspective, because of its similarity to softmax regression. (It is, however, possible to interpret the output of a multiplayer perceptron non-probabilistically, to use it for regression rather than classification, and to train it by optimizing functions other than the log likelihood).

Everything we described above is still relevant to the MLP. However, there is one more fact about softmax regression that does not apply to the MLP. Specifically, softmax regression assumes that

$$p(y|x) = \frac{exp(x^T W + b)}{\sum_i exp(x^T W + b)_i} = \text{softmax}(x^T W + b).$$

The MLP makes a different assumption about the functional form of $p(y|x)$.

2.2 The multilayer perceptron model

The multilayer perceptron model assumption is very weak. Essentially, the assumption is that the relationship between inputs and outputs can be represented by the composition of several simpler functions. Each function being composed can be thought of as another “layer” or stage of processing. The number of compositions determines the “depth” of the model.

Suppose we have a sequence of functions implementing the layers, $g_1, g_2, \ldots, g_L$. Then the output of our MLP is

$$f(x) = g_L(g_{L-1}(\ldots g_2(g_1(x)) \ldots)).$$

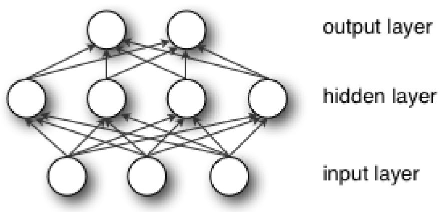

Figure 1: An MLP (or Artificial Neural Network - ANN) with a single hidden layer

In the first example we will use just two layers. The final layer will be

$$g_2(g_1) = \text{softmax}(g_1^T W^{(2)} + b^{(2)})$$

so we can think of this model as using g_1 to transform x into a different space, then doing softmax regression in that space. The vector $h(x) = g_1(x)$ constitutes the *hidden layer*. For this layer, we will use an affine transform followed by elementwise-application of the logistic sigmoid function, $\sigma(z) = \frac{1}{1+exp(-z)}$. This is a very commonly used type of layer in multilayer perceptrons. Putting it all together, we get

$$g_1(x) = \sigma(x^T W^{(1)} + b^{(1)}).$$

$W^{(1)} \in \mathbb{R}^{D \times D_h}$ is the weight matrix connecting the input vector x to the hidden layer. Each column $W_{\cdot i}^{(1)}$ represents the weights from the input units to the i-th hidden unit. Typical choices for nonlinear elementwise function also include $\tanh(x)$ which typically yields to faster training (and sometimes also to better local minima). Both the sigmoid and tanh are scalar-to-scalar functions but their natural extension to vectors and tensors consists in applying them element-wise (e.g. separately on each element of the vector, yielding a same-size vector).

The full model is thus

$$f(x) = \text{softmax}\left(\sigma(x^T W^{(1)} + b^{(1)})^T W^{(2)} + b^{(2)}\right).$$

An MLP with a single hidden layer can be represented graphically as in Fig. 1.

If we interpret $f(x)$ as defining $p(y|x)$, it makes sense to train the parameters $W^{(1)}$, $W^{(2)}$, $b^{(1)}$, and $b^{(2)}$ by maximizing the log likelihood of the training data. For MLPs, we use Stochastic Gradient Descent with minibatches. Obtaining the gradients can be achieved through the backpropagation algorithm (a special case of the chain-rule of derivation). Thankfully, since Theano performs automatic differentation, we will not need to cover this in the tutorial.

2.3 Some beneficial properties of MLPs

An obvious problem with softmax regression and other linear classifiers is that linear functions are very simple. They prevent solutions to even very simple classification problems, such as the class of 2 bit patterns whose XOR is true. XOR is true when $x = [1, 0]$ or $x = [0, 1]$ but not when $x = [0, 0]$ or $x = [1, 1]$. Suppose we draw a line that separates $[0, 0]$ from $[0, 1]$. Then it must pass through some point $[0, p]$. We require that this line also pass through $[q, 1]$ in order to separate $[0, 1]$ from $[1, 1]$. But this means it slope must be negative and its x-intercept must be negative. Since a line only has one x intercept, it does not pass between $[0, 0]$ and $[1, 0]$. Those two points belong to different classes, so any linear classifier must fail.

An MLP solves this problem by introducing extra stages of processing. In our two layer example, suppose the dimensionality of the first layer is 2. We call the outputs of this layer "hidden units" because they are neither inputs nor outputs of the system; they are unobserved variables that the network must decide what to do with. The MLP can set one of these hidden units to be active when the sum of the two input variables is less than 1. It can set the other to be active when the sum of the two input variables is greater than 1. It can then set the output unit to be active by default, and to deactivate when either of the two hidden variables is active.

More generally, an MLP with one sufficient large hidden layer can represent any function. This result is known as the "universal approximator theorem."

Another advantage of MLPs is that they can be made deeper and deeper, rather than just wider and wider. Many functions can be represented more efficiently (using fewer parameters) with a deep architecture than with a wide one. Using fewer parameters is beneficial both because the MLP takes less memory to represent, but also because the parameters may be estimated more accurately from a smaller amount of data.

2.4 Some detrimental properties of MLPs

Unfortunately, just because an MLP can represent any function does not mean that it will learn to represent the right function. The problem of overfitting can still make the MLP perform badly on the test set even if it classifies the training set perfectly. While larger MLPs are capable of fitting more complicated training sets, they are also likely to overfit worse than smaller MLPs.

A related issue with MLPs is that they have many configuration options. The model itself imposes design decisions such as what type of function to use for each layer, the dimensionality of each layer. Also, the log likelihood is no longer generally concave, so the choice of optimization procedure matters more than it did with softmax regression. These configuration options are known as "hyperparameters". Choosing the right hyperparameters is an open and exciting research problem.

2.5 Tips and Tricks for training MLPs

Source: LISA lab, University of Montreal. *DeepLearning 0.1 Documentation: Multilayer Perceptron.* `http://deeplearning.net/tutorial/mlp.html`

There are several hyper-parameters in the above code, which are not (and, generally speaking, cannot be) optimized by gradient descent. Strictly speaking, finding an optimal set of values for these hyper-parameters is not a feasible problem. First, we can't simply optimize each of them independently. Second, we cannot readily apply gradient techniques that we described previously (partly because some parameters are discrete values and others are real-valued). Third, the optimization problem is not convex and finding a (local) minimum would involve a non-trivial amount of work.

The good news is that over the last 25 years, researchers have devised various rules of thumb for choosing hyper-parameters in a neural network. A very good overview of these tricks can be found in Efficient BackProp by Yann LeCun, Leon Bottou, Genevieve Orr, and Klaus-Robert Mueller. In here, we summarize the same issues, with an emphasis on the parameters and techniques that we actually used in our code.

Nonlinearity

Two of the most common ones are the sigmoid and the tanh function. Nonlinearities that are symmetric around the origin are preferred because they tend to produce zero-mean inputs to the next layer (which is a desirable property). Empirically, we have observed that the tanh has better convergence properties.

Weight initialization

At initialization we want the weights to be small enough around the origin so that the activation function operates in its linear regime, where gradients are the largest. Other desirable properties, especially for deep networks, are to conserve variance of the activation as well as variance of back-propagated gradients from layer to layer. This allows information to flow well upward and downward in the network and reduces discrepancies between layers. Under some assumptions, a compromise between these two constraints leads to the following initialization: For tanh activation function results obtained in [4] show that the interval should be $[-\sqrt{\frac{6}{fan_{in}+fan_{out}}}, \sqrt{\frac{6}{fan_{in}+fan_{out}}}]$, where fan_{in} is the number of units in the $(i-1)$-th layer, and fan_{out} is the number of units in the i-th layer. For the sigmoid function the interval is $[-4\sqrt{\frac{6}{fan_{in}+fan_{out}}}, 4\sqrt{\frac{6}{fan_{in}+fan_{out}}}]$.

Learning rate

There is a great deal of literature on choosing a good learning rate. The simplest solution is to simply have a constant rate. Rule of thumb: try several log-spaced values $(10^{-1}, 10^{-2}, \dots)$ and narrow the (logarithmic) grid search to the region where you obtain the lowest validation error.

Decreasing the learning rate over time is sometimes a good idea. One simple rule for doing that is $\frac{\mu_0}{1+at}$ where μ_0 is the initial rate (chosen, perhaps, using the grid search technique explained above), a is a so-called "decrease constant" which controls the rate at which the learning rate decreases (typically, a smaller positive number, 10^{-2} and smaller) and t is the epoch/stage.

In Efficient BackProp by Yann LeCun, Leon Bottou, Genevieve Orr, and Klaus-Robert Muellers. authors detail procedures for choosing a learning rate for each parameter (weight) in our network and for choosing them adaptively based on the error of the classifier.

Number of hidden units

This hyper-parameter is very much dataset-dependent. Vaguely speaking, the more complicated the input distribution is, the more capacity the network will require to model it, and so the larger the number of hidden units that will be needed (note that the number of weights in a layer, perhaps a more direct measure of capacity, is $D \times D_h$ (recall D is the number of inputs and D_h is the number of hidden units).

Unless we employ some regularization scheme (early stopping or L1/L2 penalties), a typical number of hidden units vs. generalization performance graph will be U-shaped.

Regularization parameter

Typical values to try for the L1/L2 regularization parameter λ are $(10^{-2}, 10^{-3}, \dots)$. In the framework that we described so far, optimizing this parameter will not lead to significantly better solutions, but is worth exploring nonetheless.

3 Stochastic Gradient Descent

Source: LISA lab, University of Montreal. *DeepLearning 0.1 Documentation: Getting Started.* `http://deeplearning.net/tutorial/gettingstarted.html`

What is ordinary gradient descent? it is a simple algorithm in which we repeatedly make small steps downward on an error surface defined by a loss function of some parameters. For the purpose of ordinary gradient descent we consider that the training data is rolled into the loss function. Then the pseudocode of this algorithm can be described as :

Algorithm 1 GRADIENT DESCENT

```
while True do
    loss = $f$(params)
    d_loss_wrt_params = ...                          ▷ compute gradient
    params -= learning_rate * d_loss_wrt_params
    if stopping condition is met then return params
    end if
end while
```

Stochastic gradient descent (SGD) works according to the same principles as ordinary gradient descent, but proceeds more quickly by estimating the gradient from just a few examples at a time instead of the entire training set. In its purest form, we estimate the gradient from just a single example at a time.

The variant that we recommend for deep learning is a further twist on stochastic gradient descent using so-called “minibatches”. Minibatch SGD works identically to SGD, except that we use more than one training example to make each estimate of the gradient. This technique reduces variance in the estimate of the gradient, and often makes better use of the hierarchical memory organization in modern computers.

There is a tradeoff in the choice of the minibatch size B. The reduction of variance and use of SIMD instructions helps most when increasing B from 1 to 2, but the marginal improvement fades rapidly to nothing. With large B, time is wasted in reducing the variance of the gradient estimator, that time would be

Algorithm 2 STOCHASTIC GRADIENT DESCENT

```
for $(x_i, y_i) \in \mathcal{D}_{train}$ do
                              ▷ imagine an infinite generator that may repeat
                              ▷ examples (if there is only a finite training set)
    loss = $f$(params, $x_i$, $y_i$)
    d_loss_wrt_params = ...                              ▷ compute gradient
    params $-=$ learning_rate * d_loss_wrt_params
    if stopping condition is met then return params
    end if
end for
```

Algorithm 3 MINIBATCH SGD

```
for (x_batch,y_batch) $\in$ train_batches do
                                           ▷ imagine an infinite generator
                                           ▷ that may repeat examples
    loss = $f$(params, x_batch, y_batch)
    d_loss_wrt_params = ...                              ▷ compute gradient
    params $-=$ learning_rate * d_loss_wrt_params
    if stopping condition is met then return params
    end if
end for
```

better spent on additional gradient steps. An optimal B is model-, dataset-, and hardware-dependent, and can be anywhere from 1 to maybe several hundreds.

Note

If you are training for a fixed number of epochs, the minibatch size becomes important because it controls the number of updates done to your parameters. Training the same model for 10 epochs using a batch size of 1 yields completely different results compared to training for the same 10 epochs but with a batchsize of 20. Keep this in mind when switching between batch sizes and be prepared to tweak all the other parameters acording to the batch size used.

4 Regularization

Source: LISA lab, University of Montreal. *DeepLearning 0.1 Documentation: Getting Started.* `http://deeplearning.net/tutorial/gettingstarted.html`

There is more to machine learning than optimization. When we train our model from data we are trying to prepare it to do well on new examples, not the ones it has already seen. The training loop above for MSGD does not take this into account, and may overfit the training examples. A way to combat overfitting is through *regularization.* There are several techniques for regularization; the ones we will explain here are L1/L2 regularization and early-stopping.

4.1 L1 and L2 regularization

L1 and L2 regularization involve adding an extra term to the loss function, which penalizes certain parameter configurations. Formally, if our loss function is:

$$NLL(\theta, \mathcal{D}) = -\sum_{i=0}^{|\mathcal{D}|} \log P(Y = y^{(i)} | x^{(i)}, \theta)$$

then the regularized loss will be:

$$E(\theta, \mathcal{D}) = NLL(\theta, \mathcal{D}) + \lambda R(\theta)$$

or, in our case

$$E(\theta, \mathcal{D}) = NLL(\theta, \mathcal{D}) + \lambda \|\theta\|_p^p$$

where

$$\|\theta\|_p = \left(\sum_{j=0} |\theta_j|^p \right)^{\frac{1}{p}}$$

which is the L_p norm of θ. λ is a hyper-parameter which controls the relative importance of the regularization parameter. Commonly used values for p are 1 and 2, hence the L1/L2 nomenclature. If $p = 2$, then the regularizer is also called "weight decay".

In principle, adding a regularization term to the loss will encourage smooth network mappings in a neural network (by penalizing large values of the parameters, which decreases the amount of nonlinearity that the network models). More intuitively, the two terms (NLL and $R(\theta)$) correspond to modelling the data well (NLL) and having "simple" or "smooth" solutions ($R(\theta)$). Thus, minimizing the sum of both will, in theory, correspond to finding the right trade-off between the fit to the training data and the "generality" of the solution that is found. To follow Occam's razor principle, this minimization should find us the simplest solution (as measured by our simplicity criterion) that fits the training data.

Note that the fact that a solution is "simple" does not mean that it will generalize well. Empirically, it was found that performing such regularization in the context of neural networks helps with generalization, especially on small datasets.

4.2 Early-Stopping

Early-stopping combats overfitting by monitoring the model's performance on a validation set. A validation set is a set of examples that we never use for gradient descent, but which is also not a part of the test set. The validation examples are considered to be representative of future test examples. We can use them during training because they are not part of the test set. If the model's performance ceases to improve sufficiently on the validation set, or even degrades with further optimization, then the heuristic implemented here gives up on much further optimization.

5 Convolutional Neural Networks

Source: LISA lab, University of Montreal. *DeepLearning 0.1 Documentation: Convolutional Neural Networks (LeNet).* `http://deeplearning.net/tutorial/lenet.html`

5.1 Motivation

Convolutional Neural Networks (CNN) are variants of MLPs which are inspired from biology. From Hubel and Wiesel's early work on the cat's visual cortex [7], we know there exists a complex arrangement of cells within the visual cortex. These cells are sensitive to small sub-regions of the input space, called a receptive field, and are tiled in such a way as to cover the entire visual field. These filters are local in input space and are thus better suited to exploit the strong spatially local correlation present in natural images.

Additionally, two basic cell types have been identified: simple cells (S) and complex cells (C). Simple cells (S) respond maximally to specific edge-like stimulus patterns within their receptive field. Complex cells (C) have larger receptive fields and are locally invariant to the exact position of the stimulus.

The visual cortex being the most powerful "vision" system in existence, it seems natural to emulate its behavior. Many such neurally inspired models can be found in the litterature. To name a few: the NeoCognitron [8], HMAX [9] and LeNet-5 [10], which will be the focus of this tutorial.

5.2 Sparse Connectivity

CNNs exploit spatially local correlation by enforcing a local connectivity pattern between neurons of adjacent layers. The input hidden units in the m-th layer are connected to a local subset of units in the $(m-1)$-th layer, which have spatially contiguous receptive fields. We can illustrate this graphically as follows (Fig. 2).

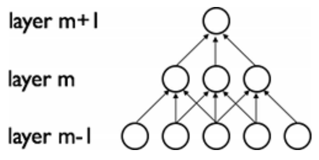

Figure 2: CNN's sparse connectivity

Imagine that layer $m-1$ is the input retina. In the above, units in layer m have receptive fields of width 3 with respect to the input retina and are thus only connected to 3 adjacent neurons in the layer below (the retina). Units in layer m have a similar connectivity with the layer below. We say that their receptive field with respect to the layer below is also 3, but their receptive field with respect to the input is larger (it is 5). The architecture thus confines the learnt "filters" (corresponding to the input producing the strongest response) to be a spatially local pattern (since each unit is unresponsive to variations outside of its receptive field with respect to the retina). As shown above, stacking many such layers leads to "filters" (not anymore linear) which become increasingly "global" however (i.e spanning a larger region of pixel space). For example, the unit in hidden layer $m+1$ can encode a non-linear feature of width 5 (in terms of pixel space).

5.3 Shared Weights

In CNNs, each sparse filter h_i is additionally replicated across the entire visual field. These "replicated" units form a *feature map*, which share the same parametrization, i.e. the same weight vector and the same bias. In Fig. 3, we

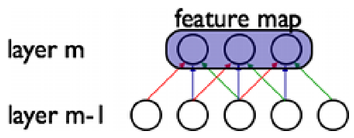

Figure 3: Hidden units belonging to the same feature map. Weights of the same color are shared, i.e. are constrained to be identical.

show 3 hidden units belonging to the same feature map. Weights of the same color are shared, i.e. are constrained to be identical. Gradient descent can still be used to learn such shared parameters, and requires only a small change to the original algorithm. The gradient of a shared weight is simply the sum of the gradients of the parameters being shared.

Why are shared weights interesting? Replicating units in this way allows for features to be detected regardless of their position in the visual field. Additionally, weight sharing offers a very efficient way to do this, since it greatly reduces the number of free parameters to learn. By controlling model capacity, CNNs tend to achieve better generalization on vision problems.

5.4 Details and Notation

Conceptually, a feature map is obtained by convolving the input image with a linear filter, adding a bias term and then applying a non-linear function. If we denote the k-th feature map at a given layer as h^k, whose filters are determined by the weights W^k and bias b_k, then the feature map h^k is obtained as follows (for tanh non-linearities):

$$h^k_{ij} = \tanh((W^k * x)_{ij} + b_k).$$

Note
Recall the following definition of convolution for a 1D signal.

$$o[n] = f[n] * g[n] = \sum_{u=-\infty}^{\infty} f[u]g[u-n] = \sum_{u=-\infty}^{\infty} f[n-u]g[u].$$

This can be extended to 2D as follows:

$$o[m,n] = f[m,n] * g[m,n] = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f[u,v]g[u-m,v-n].$$

To form a richer representation of the data, hidden layers are composed of a set of multiple feature maps, $\{h^{(k)}, k = 0..K\}$. The weights W of this layer can be parametrized as a 4D tensor (destination feature map index, source feature map index, source vertical position index, source horizontal position index) and the biases b as a vector (one element per destination feature map index). We illustrate this graphically as follows:

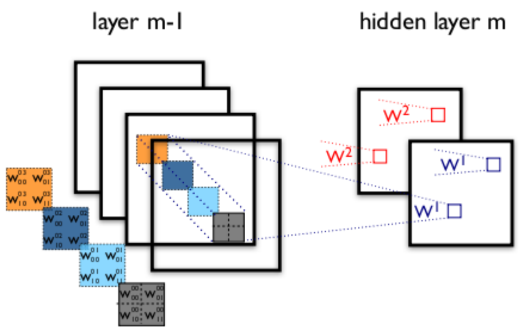

Figure 4: Example of a convolutional layer

Here, we show two layers of a CNN, containing 4 feature maps at layer $(m-1)$ and 2 feature maps (h^0 and h^1) at layer m. Pixels (neuron outputs) in h^0 and h^1 (outlined as blue and red squares) are computed from pixels of layer $(m-1)$ which fall within their 2x2 receptive field in the layer below (shown as colored rectangles). Notice how the receptive field spans all four input feature maps. The weights W^0 and W^1 of h^0 and h^1 are thus 3D weight tensors. The leading dimension indexes the input feature maps, while the other two refer to the pixel coordinates.

Putting it all together, W_{ij}^{kl} denotes the weight connecting each pixel of the k-th feature map at layer m, with the pixel at coordinates (i, j) of the l-th feature map of layer $(m-1)$.

5.5 MaxPooling

Another important concept of CNNs is that of max-pooling, which is a form of non-linear down-sampling. Max-pooling partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum value.

Max-pooling is useful in vision for two reasons: (1) it reduces the computational complexity for upper layers and (2) it provides a form of translation invariance. To understand the invariance argument, imagine cascading a max-pooling layer with a convolutional layer. There are 8 directions in which one can translate the input image by a single pixel. If max-pooling is done over a 2x2 region, 3 out of these 8 possible configurations will produce exactly the same output at the convolutional layer. For max-pooling over a 3x3 window, this jumps to 5/8.

Since it provides additional robustness to position, max-pooling is thus a "smart" way of reducing the dimensionality of intermediate representations.

5.6 The Full Model: LeNet

Sparse, convolutional layers and max-pooling are at the heart of the LeNet family of models. While the exact details of the model will vary greatly, the figure below shows a graphical depiction of a LeNet model.

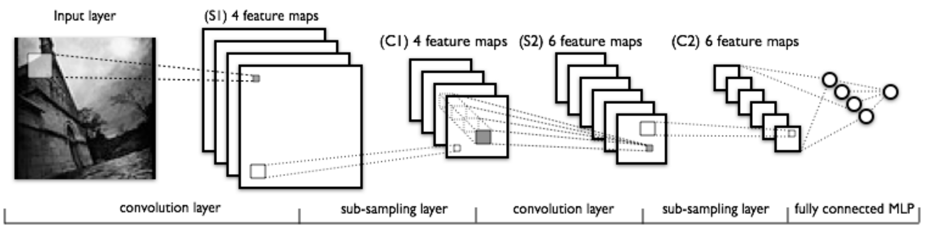

Figure 5: LeNet model

The lower-layers are composed to alternating convolution and max-pooling layers. The upper-layers however are fully-connected and correspond to a traditional MLP (hidden layer + logistic regression). The input to the first fully-connected layer is the set of all features maps at the layer below.

From an implementation point of view, this means lower-layers operate on 4D tensors. These are then flattened to a 2D matrix of rasterized feature maps, to be compatible with our previous MLP implementation.

5.7 Tips and Tricks

Choosing Hyperparameters

CNNs are especially tricky to train, as they add even more hyper-parameters than a standard MLP. While the usual rules of thumb for learning rates and regularization constants still apply, the following should be kept in mind when optimizing CNNs.

Number of filters

When choosing the number of filters per layer, keep in mind that computing the activations of a single convolutional filter is much more expensive than with traditional MLPs!

Assume layer $(l-1)$ contains K^{l-1} feature maps and $M \times N$ pixel positions (i.e., number of positions times number of feature maps), and there are K^l filters at layer l of shape $m \times n$. Then computing a feature map (applying an $m \times n$ filter at all $(M-m) \times (N-n)$ pixel positions where the filter can be applied) costs $(M-m) \times (N-n) \times m \times n \times K^{l-1}$. The total cost is K^l times that. Things may be more complicated if not all features at one level are connected to all features at the previous one.

For a standard MLP, the cost would only be $K^l \times K^{l-1}$ where there are K^l different neurons at level l. As such, the number of filters used in CNNs is typically much smaller than the number of hidden units in MLPs and depends on the size of the feature maps (itself a function of input image size and filter shapes).

Since feature map size decreases with depth, layers near the input layer will tend to have fewer filters while layers higher up can have much more. In fact, to equalize computation at each layer, the product of the number of features and the number of pixel positions is typically picked to be roughly constant across layers. To preserve the information about the input would require keeping the total number of activations (number of feature maps times number of pixel positions) to be non-decreasing from one layer to the next (of course we could hope to get away with less when we are doing supervised learning). The number of feature maps directly controls capacity and so that depends on the number of available examples and the complexity of the task.

Filter Shape

Common filter shapes found in the litterature vary greatly, usually based on the dataset. Best results on MNIST-sized images (28x28) are usually in the 5x5 range on the first layer, while natural image datasets (often with hundreds of pixels in each dimension) tend to use larger first-layer filters of shape 12x12 or 15x15.

The trick is thus to find the right level of "granularity" (i.e. filter shapes) in order to create abstractions at the proper scale, given a particular dataset.

Max Pooling Shape

Typical values are 2x2 or no max-pooling. Very large input images may warrant 4x4 pooling in the lower-layers. Keep in mind however, that this will reduce the dimension of the signal by a factor of 16, and may result in throwing away too much information.

6 Denoising Autoencoder

Source: LISA lab, University of Montreal. *DeepLearning 0.1 Documentation: Denoising Autoencoders.* `http://deeplearning.net/tutorial/dA.html`

The Denoising Autoencoder (dA) is an extension of a classical autoencoder and it was introduced as a building block for deep networks in [12]. We will start the tutorial with a short discussion on Auto-Encoders.

6.1 Auto-Encoders

See section 4.6 of [13] for an overview of auto-encoders. An autoencoder takes an input $x \in [0,1]^d$ and first maps it (with an *encoder*) to a hidden representation $y \in [0,1]^{d'}$ through a deterministic mapping, e.g.:

$$y = s(Wx + b)$$

Where s is a non-linearity such as the sigmoid. The latent representation y, or *code* is then mapped back (with a *decoder*) into a *reconstruction* z of same shape as x through a similar transformation, e.g.:

$$z = s(W'y + b')$$

where $'$ does not indicate transpose, and z should be seen as a prediction of x, given the code y. The weight matrix W' of the reverse mapping may be optionally constrained by $W' = W^T$, which is an instance of *tied weights.* The parameters of this model (namely W, b, b' and, if one doesn't use tied weights, also W') are optimized such that the average reconstruction error is minimized. The reconstruction error can be measured in many ways, depending on the appropriate distributional assumptions on the input given the code, e.g., using the traditional squared error $L(x, z) = \|x - z\|^2$, or if the input is interpreted as either bit vectors or vectors of bit probabilities by the reconstruction cross-entropy defined as :

$$L_H(x, z) = -\sum_{k=1}^{d}[x_k \log z_k + (1 - x_k)\log(1 - z_k)]$$

The hope is that the code is a distributed representation that captures the coordinates along the main factors of variation in the data (similarly to how the projection on principal components captures the main factors of variation in the data). Because is viewed as a lossy compression of x, it cannot be a good compression (with small loss) for all x, so learning drives it to be one that is a good compression in particular for training examples, and hopefully for others as well, but not for arbitrary inputs. That is the sense in which an auto-encoder generalizes: it gives low reconstruction error to test examples from the same distribution as the training examples, but generally high reconstruction error to uniformly chosen configurations of the input vector.

If there is one linear hidden layer (the code) and the mean squared error criterion is used to train the network, then the k hidden units learn to project the input in the span of the first k principal components of the data. If the hidden layer is non-linear, the auto-encoder behaves differently from PCA, with the ability to capture multi-modal aspects of the input distribution. The departure from PCA becomes even more important when we consider *stacking multiple encoders* (and their corresponding decoders) when building a deep auto-encoder [14].

One serious potential issue with auto-encoders is that if there is no other constraint besides minimizing the reconstruction error, then an auto-encoder with n inputs and an encoding of dimension at least n could potentially just learn the identity function, for which many encodings would be useless (e.g., just copying the input), i.e., the autoencoder would not differentiate test examples (from the training distribution) from other input configurations. Surprisingly, experiments reported in [15] nonetheless suggest that in practice, when trained with stochastic gradient descent, non-linear auto-encoders with more hidden units than inputs (called overcomplete) yield useful representations (in the sense of classification error measured on a network taking this representation in input). A simple explanation is based on the observation that stochastic gradient descent with early stopping is similar to an L2 regularization of the parameters. To achieve perfect reconstruction of continuous inputs, a one-hidden layer auto-encoder with non-linear hidden units needs very small weights in the first (encoding) layer (to bring the non-linearity of the hidden units in their linear regime) and very large weights in the second (decoding) layer. With binary inputs, very large weights are also needed to completely minimize the reconstruction error. Since the implicit or explicit regularization makes it difficult to reach large-weight solutions, the optimization algorithm finds encodings which only work well for examples similar to those in the training set, which is what we want. It means that the representation is exploiting statistical regularities present in the training set, rather than learning to replicate the identity function.

There are different ways that an auto-encoder with more hidden units than inputs could be prevented from learning the identity, and still capture something useful about the input in its hidden representation. One is the addition of sparsity (forcing many of the hidden units to be zero or near-zero), and it has been exploited very successfully by many [16, 17]. Another is to add randomness in the transformation from input to reconstruction. This is exploited in *Restricted Boltzmann Machines* (discussed later), as well as in *Denoising Auto-Encoders*, discussed below.

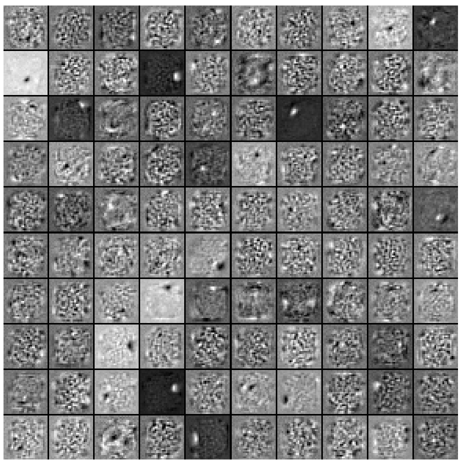

Figure 6: Auto-encoder filters

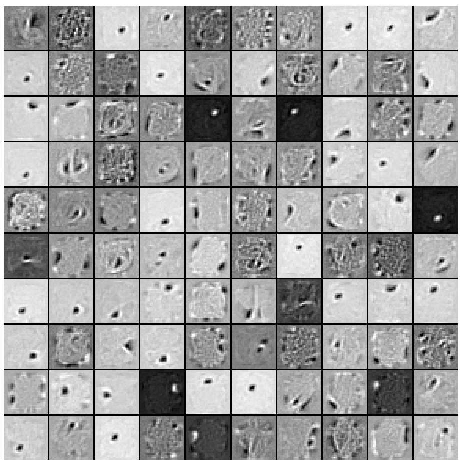

Figure 7: Denoising Auto-encoder filters

6.2 Denoising Auto-Encoders

The idea behind denoising auto-encoders is simple. In order to force the hidden layer to discover more robust features and prevent it from simply learning the identity, we train the autoencoder to reconstruct the input from a corrupted version of it.

The denoising auto-encoder is a stochastic version of the auto-encoder. Intuitively, a denoising auto-encoder does two things: try to encode the input (preserve the information about the input), and try to undo the effect of a corruption process stochastically applied to the input of the auto-encoder. The latter can only be done by capturing the statistical dependencies between the inputs. The denoising auto-encoder can be understood from different perspectives (the manifold learning perspective, stochastic operator perspective, bottom-up – information theoretic perspective, top-down – generative model perspective), all of which are explained in [12]. See also section 7.2 of [13] for an overview of auto-encoders.

In [12], the stochastic corruption process consists in randomly setting some of the inputs (as many as half of them) to zero. Hence the denoising auto-encoder is trying to predict the corrupted (i.e. missing) values from the uncorrupted (i.e., non-missing) values, for randomly selected subsets of missing patterns. Note how being able to predict any subset of variables from the rest is a sufficient condition for completely capturing the joint distribution between a set of variables (this is how *Gibbs sampling* works).

6.3 Applying Denoising Auto-Encoders to MNIST dataset

In [11] the reader can find a Theano code for applying auto-encoders and denoising auto-encoders to the MNIST dataset.

The resulted filters when we do not use any noise are shown in Fig. 6. The filters for 30 percent noise are shown in Fig. 7.

7 Stacked Denoising Autoencoders

The Stacked Denoising Autoencoder (SdA) is an extension of the stacked autoencoder [15] and it was introduced in [12].

7.1 Stacked Autoencoders

The denoising autoencoders can be stacked to form a deep network by feeding the latent representation (output code) of the denoising auto-encoder found on the layer below as input to the current layer. The unsupervised pre-training of such an architecture is done one layer at a time. Each layer is trained as a denoising auto-encoder by minimizing the reconstruction of its input (which is the output code of the previous layer). Once the first layers are trained, we can train the -th layer because we can now compute the code or latent representation from the layer below. Once all layers are pre-trained, the network goes through a second stage of training called fine-tuning. Here we consider supervised fine-tuning where we want to minimize prediction error on a supervised task. For this we first add a logistic regression layer on top of the network (more precisely on the output code of the output layer). We then train the entire network as we would train a multilayer perceptron. At this point, we only consider the encoding parts of each auto-encoder. This stage is supervised, since now we use the target class during training.

8 Restricted Boltzmann Machines (RBM)

Source: LISA lab, University of Montreal. *DeepLearning 0.1 Documentation: Restricted Boltzmann Machines (RBM).* `http://deeplearning.net/tutorial/rbm.html`

8.1 Energy-Based Models (EBM)

Energy-based models associate a scalar energy to each configuration of the variables of interest. Learning corresponds to modifying that energy function so that its shape has desirable properties. For example, we would like plausible or desirable configurations to have low energy. Energy-based probabilistic models define a probability distribution through an energy function, as follows:

$$p(x) = \frac{e^{-E(x)}}{Z} \tag{1}$$

The normalizing factor is called the partition function by analogy with physical systems.

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

An energy-based model can be learnt by performing (stochastic) gradient descent on the empirical negative log-likelihood of the training data. As for the logistic regression we will first define the log-likelihood and then the loss function as being the negative log-likelihood.

$$L(\theta, \mathcal{D}) = \frac{1}{N} \sum_{x^{(i)} \in \mathcal{D}} \log p(x^{(i)})$$

$$\ell(\theta, \mathcal{D}) = -L(\theta, \mathcal{D})$$

using the stochastic gradient $-\frac{\partial \log p(x^{(i)})}{\partial \theta}$, where θ are the parameters of the model.

8.2 EBMs with Hidden Units

In many cases of interest, we do not observe the example x fully, or we want to introduce some non-observed variables to increase the expressive power of the model. So we consider an observed part (still denoted x here) and a *hidden* part h. We can then write:

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

In such cases, to map this formulation to one similar to (1), we introduce the notation (inspired from physics) of *free energy*, defined as follows:

$$\mathcal{F}(x) = -\log \sum_h e^{-E(x,h)}$$

which allows us to write,

$$P(x) = \frac{e^{-\mathcal{F}(x)}}{Z} \text{ whith } Z = \sum_x e^{-\mathcal{F}(x)}$$

The data negative log-likelihood gradient then has a particularly interesting form:

$$-\frac{\partial \log p(x)}{\partial \theta} = \frac{\partial \mathcal{F}(x)}{\partial \theta} - \sum_{\tilde{x}} p(\tilde{x}) \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta} \tag{2}$$

Notice that the above gradient contains two terms, which are referred to as the *positive* and *negative phase*. The terms positive and negative do not refer to the sign of each term in the equation, but rather reflect their effect on the probability density defined by the model. The first term increases the probability of training data (by reducing the corresponding free energy), while the second term decreases the probability of samples generated by the model.

It is usually difficult to determine this gradient analytically, as it involves the computation of $E_P\left[\frac{\partial \mathcal{F}(x)}{\partial \theta}\right]$. This is nothing less than an expectation over all possible configurations of the input x (under the distribution P formed by the model).

The first step in making this computation tractable is to estimate the expectation using a fixed number of model samples. Samples used to estimate the negative phase gradient are referred to as *negative particles*, which are denoted as $\mathcal{N}$. The gradient can then be written as:

$$-\frac{\partial \log p(x)}{\partial \theta} \approx \frac{\partial \mathcal{F}(x)}{\partial \theta} - \frac{1}{|\mathcal{N}|} \sum_{\tilde{x} \in \mathcal{N}} \frac{\partial \mathcal{F}(\tilde{x})}{\partial \theta} \tag{3}$$

where we would ideally like elements $\tilde{x}$ of $\mathcal{N}$ to be sampled according to P (i.e. we are doing Monte-Carlo). With the above formula, we almost have a pratical, stochastic algorithm for learning an EBM. The only missing ingredient is how to extract these negative particles $\mathcal{N}$. While the statistical literature abounds with sampling methods, Markov Chain Monte Carlo methods are especially well suited for models such as the Restricted Boltzmann Machines (RBM), a specific type of EBM.

8.3 Restricted Boltzmann Machines (RBM)

Boltzmann Machines (BMs) are a particular form of log-linear Markov Random Field (MRF), i.e., for which the energy function is linear in its free parameters. To make them powerful enough to represent complicated distributions (i.e., go from the limited parametric setting to a non-parametric one), we consider that some of the variables are never observed (they are called hidden). By having more hidden variables (also called hidden units), we can increase the modeling capacity of the Boltzmann Machine (BM). Restricted Boltzmann Machines further restrict BMs to those without visible-visible and hidden-hidden connections. A graphical depiction of an RBM is shown in Fig. 8.

The energy function of an RBM is defined as:

$$E(x, h) = -b^T x - c^T x - h^T W x \tag{4}$$

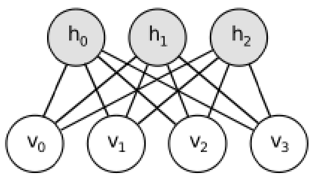

Figure 8: Restricted Boltzmann Machine (RBM)

where W represents the weights connecting hidden and visible units and b and c are the offsets of the visible and hidden layers respectively.

This translates directly to the following free energy formula:

$$\mathcal{F}(x) = -b^T x - \sum_i \log \sum_{h_i} e^{h_i(c_i + W_i x)}.$$

Because of the specific structure of RBMs, visible and hidden units are conditionally independent given one-another. Using this property, we can write:

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

$$p(x|h) = \prod_i p(x_i|h).$$

8.4 RBMs with binary units

In the commonly studied case of using binary units (where x_j and $h_i \in \{0,1\}$), we obtain from Eq. (4) and (1), a probabilistic version of the usual neuron activation function:

$$\begin{aligned} P(h_i = 1|x) &= \text{sigmoid}(c_i + W_i x) \\ P(x_j = 1|h) &= \text{sigmoid}(b_j + W_j^T x) \end{aligned}$$

The free energy of an RBM with binary units further simplifies to:

$$\mathcal{F}(x) = -b^T x - \sum_i \log(1 + e^{c_i + W_i x}). \tag{5}$$

Update Equations with Binary Units

Combining Eqs. (3) with (5), we obtain the following log-likelihood gradients for an RBM with binary units:

$$\begin{aligned} -\frac{\partial \log p(x)}{\partial W_{ij}} &= E_x[p(h_i|x) \cdot x_j] - x_j^{(i)} \cdot \text{sigmoid}(W_i \cdot x^{(i)} + c_i) \\ -\frac{\partial \log p(x)}{\partial c_i} &= E_x[p(h_i|x)] - \text{sigmoid}(W_i \cdot x^{(i)}) \\ -\frac{\partial \log p(x)}{\partial b_j} &= E_x[p(x_j|h)] - x_j^{(i)} \end{aligned}$$

For a more detailed derivation of these equations, we refer the reader to [19], or to section 5 of [13]. We will however not use these formulas, but rather get the gradient using Theano T.grad from equation (2).

8.5 Sampling in an RBM

Samples of $p(x)$ can be obtained by running a Markov chain to convergence, using Gibbs sampling as the transition operator.

Gibbs sampling of the joint of N random variables $S = (S_1, \ldots, S_N)$ is done through a sequence of N sampling sub-steps of the form $S_i \sim p(S_i|S_{-i})$ where S_{-i} contains the $N-1$ other random variables in S excluding S_i.

For RBMs, S consists of the set of visible and hidden units. However, since they are conditionally independent, one can perform block Gibbs sampling. In this setting, visible units are sampled simultaneously given fixed values of the hidden units. Similarly, hidden units are sampled simultaneously given the visibles. A step in the Markov chain is thus taken as follows:

$$\begin{aligned} h^{(n+1)} &\sim \text{sigmoid}(W^T x^{(n)} + c) \\ x^{(n+1)} &\sim \text{sigmoid}(W h^{(n+1)} + b), \end{aligned}$$

where $h^{(n)}$ refers to the set of all hidden units at the n-th step of the Markov chain. What it means is that, for example, $h_i^{(n+1)}$ is randomly chosen to be 1 (versus 0) with probability $\text{sigmoid}(W_i^T x^{(n)} + c_i)$, and similarly, $x_j^{(n+1)}$ is randomly chosen to be 1 (versus 0) with probability $\text{sigmoid}(W_{.j} h^{(n+1)} + b_j)$.

This can be illustrated graphically:

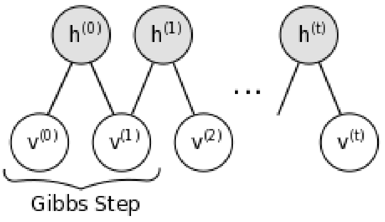

Figure 9: Markov Chain for a Restricted Boltzmann Machine

As $t \to \infty$, samples $(x^{(t)}, h^{(t)})$ are guaranteed to be accurate samples of $p(x,h)$.

In theory, each parameter update in the learning process would require running one such chain to convergence. It is needless to say that doing so would be prohibitively expensive. As such, several algorithms have been devised for RBMs, in order to efficiently sample from $p(x,h)$ during the learning process.

Contrastive Divergence (CD-k)

Contrastive Divergence uses two tricks to speed up the sampling process:

- since we eventually want $p(x) \approx p_t rain(x)$ (the true, underlying distribution of the data), we initialize the Markov chain with a training example (i.e., from a distribution that is expected to be close to p, so that the chain will be already close to having converged to its final distribution p).

- CD does not wait for the chain to converge. Samples are obtained after only k-steps of Gibbs sampling. In pratice, has been shown to work surprisingly well.

Persistent CD

Persistent CD [20] uses another approximation for sampling from $p(x, h)$. It relies on a single Markov chain, which has a persistent state (i.e., not restarting a chain for each observed example). For each parameter update, we extract new samples by simply running the chain for k-steps. The state of the chain is then preserved for subsequent updates.

The general intuition is that if parameter updates are small enough compared to the mixing rate of the chain, the Markov chain should be able to "catch up" the changes in the model.

9 Deep Belief Networks

Source: LISA lab, University of Montreal. *DeepLearning 0.1 Documentation: Deep Belief Networks.* `http://deeplearning.net/tutorial/DBN.html`

9.1 Deep Belief Networks

Hinton and Salakhutdinov [14] showed that RBMs can be stacked and trained in a greedy manner to form so-called Deep Belief Networks (DBN). DBNs are graphical models which learn to extract a deep hierarchical representation of the training data. They model the joint distribution between observed vector x and the ℓ hidden layers h^k as follows:

$$P(x, h^1, \ldots, h^\ell) = \left(\prod_{k=0}^{\ell-2} P(h^k|h^{k+1})\right) P(h^{\ell-1}|h^\ell)$$

where $x = h^0$, $P(h^{k-1}|h^k)$ is a conditional distribution for the visible units conditioned on the hidden units of the RBM at level k, and $P(h^{\ell-1}|h^\ell)$ is the visible-hidden joint distribution in the top-level RBM. This is illustrated in Fig. 10.

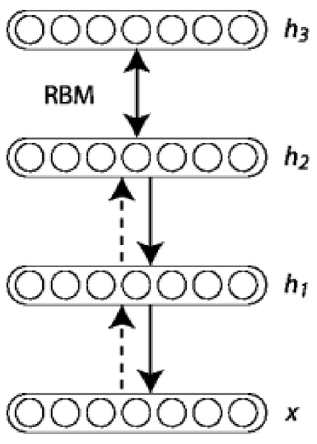

Figure 10: A Deep Belief Network

The principle of greedy layer-wise unsupervised training can be applied to

DBNs with RBMs as the building blocks for each layer [14, 15]. The process is as follows:

1. Train the first layer as an RBM that models the raw input $x = h^{(0)}$ as its visible layer.
2. Use that first layer to obtain a representation of the input that will be used as data for the second layer. Two common solutions exist. This representation can be chosen as being the mean activations $p(h^{(1)} = 1|h^{(0)})$ or samples of $p(h^{(1)}|h^{(0)})$.
3. Train the second layer as an RBM, taking the transformed data (samples or mean activations) as training examples (for the visible layer of that RBM).
4. Iterate (2 and 3) for the desired number of layers, each time propagating upward either samples or mean values.
5. Fine-tune all the parameters of this deep architecture with respect to a proxy for the DBN log-likelihood, or with respect to a supervised training criterion (after adding extra learning machinery to convert the learned representation into supervised predictions, e.g. a linear classifier).

In the online tutorial [21], the focus is on fine-tuning via supervised gradient descent. Specifically, a logistic regression classifier is implemented to classify the input x based on the output of the last hidden layer $h^{(\ell)}$ of the DBN. Fine-tuning is then performed via supervised gradient descent of the negative log-likelihood cost function. Since the supervised gradient is only non-null for the weights and hidden layer biases of each layer (i.e. null for the visible biases of each RBM), this procedure is equivalent to initializing the parameters of a deep MLP with the weights and hidden layer biases obtained with the unsupervised training strategy.

References

[1] Ian Goodfellow. *“Pylearn2 Tutorial: Softmax regression”*. `http://nbviewer.ipython.org/github/lisa-lab/pylearn2/blob/master/pylearn2/scripts/tutorials/softmax_regression/softmax_regression.ipynb`.

[2] Ian Goodfellow. *“Pylearn2 Tutorial: Multilayer Perceptron”*. `http://nbviewer.ipython.org/github/lisa-lab/pylearn2/blob/master/pylearn2/scripts/tutorials/multilayer_perceptron/multilayer_perceptron.ipynb`.

[3] LISA lab, University of Montreal. *DeepLearning 0.1 Documentation: Multilayer Perceptron.* `http://deeplearning.net/tutorial/mlp.html`.

[4] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[5] LISA lab, University of Montreal. *DeepLearning 0.1 Documentation: Getting Started*. `http://deeplearning.net/tutorial/gettingstarted.html`.

[6] LISA lab, University of Montreal. *DeepLearning 0.1 Documentation: Convolutional Neural Networks (LeNet)*. `http://deeplearning.net/tutorial/lenet.html`.

[7] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

[8] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[9] Thomas Serre, Lior Wolf, Stanley Bileschi, Maximilian Riesenhuber, and Tomaso Poggio. Robust object recognition with cortex-like mechanisms. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(3):411–426, 2007.

[10] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[11] LISA lab, University of Montreal. *DeepLearning 0.1 Documentation: Denoising Autoencoders*. `http://deeplearning.net/tutorial/dA.html`.

[12] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

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

[14] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

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

[16] Christopher Poultney Marc'Aurelio Ranzato, Sumit Chopra, and Yann LeCun. Efficient learning of sparse representations with an energy-based model. In *Proceedings of NIPS*, 2007.

[17] Honglak Lee, Chaitanya Ekanadham, and Andrew Y Ng. Sparse deep belief net model for visual area v2. In *Advances in neural information processing systems*, pages 873–880, 2008.

[18] LISA lab, University of Montreal. *DeepLearning 0.1 Documentation: Restricted Boltzmann Machines (RBM)*. `http://deeplearning.net/tutorial/rbm.html`.

[19] LISA lab, University of Montreal. *Contrastive divergence multi-layer RBMs*. `http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/DBNEquations`.

[20] Tijmen Tieleman. Training restricted boltzmann machines using approximations to the likelihood gradient. In *Proceedings of the 25th international conference on Machine learning*, pages 1064–1071. ACM, 2008.

[21] LISA lab, University of Montreal. *DeepLearning 0.1 Documentation: Deep Belief Networks*. `http://deeplearning.net/tutorial/DBN.html`.