RNPS No. 2142
ISSN 2072-6287
Versión Digital

CENATAV
Centro de Aplicaciones de Tecnologías de Avanzada

REPORTE TÉCNICO
Reconocimiento de Patrones

SERIE AZUL

Application of Deep Learning Methods in Speech Processing: a New Representation for Language Recognitions

Ana Montalvo Bereau and José R. Calvo de Lara

RT_082 **abril 2016**

7ma. No. 21812 e/218 y 222,
Rpto. Siboney, Playa;
La Habana.
Cuba. C.P. 12200
www.cenatav.co.cu

RNPS No. 2142
ISSN 2072-6287
Versión Digital

SERIE AZUL

CENATAV
Centro de Aplicaciones de Tecnologías de Avanzada

REPORTE TÉCNICO
Reconocimiento de Patrones

Application of Deep Learning Methods in Speech Processing: a New Representation for Language Recognition

Ana Montalvo Bereau and José R. Calvo de Lara

RT_082 **abril 2016**

7ma. No. 21812 e/218 y 222,
Rpto. Siboney, Playa;
La Habana.
Cuba. C.P. 12200
www.cenatav.co.cu

Table of content

Application of Deep Learning Methods in Speech Processing: a New Representation for Language Recognition

Ana Montalvo Bereau and José R. Calvo de Lara

Equipo de Investigaciones de Imágenes y Señales, CENATAV - DATYS, La Habana, Cuba
{amontalvo, jcalvo}@cenatav.co.cu

RT_082, Serie Azul, CENATAV
Aceptado: 16 de Abril de 2016

Abstract. This report summarizes the experiences and knowledge acquired during the training process of deep neural networks for spoken language recognition. It explains and emphasizes the most sensitive issues of the procedure, in order to facilitate the transit through such arduous adjustment process.
Keywords: deep learning, neural networks, spoken language recognition, speech recognition.

Resumen. El presente reporte resume las experiencias y conocimientos adquiridos durante el proceso de entrenamiento de redes neuronales profundas para el reconocimiento del idioma hablado. Explica y enfatiza, las cuestiones más sensibles de dicho procedimiento, con vistas a facilitar el tránsito por tan arduo proceso de ajuste.
Palabras clave: aprendizaje profundo, redes neuronales, reconocimiento del idioma hablado, reconocimiento de habla.

1 Introduction

Until recently, most machine learning and signal processing techniques had exploited shallow-structured architectures. These architectures typically contain at most one or two layers of nonlinear feature transformations. Examples of the shallow architectures are gaussian mixture models (GMMs), linear or nonlinear dynamical systems, conditional random fields, maximum entropy models, support vector machines (SVMs), logistic regression, kernel regression, multilayer perceptrons (MLPs) with a single hidden layer including extreme learning machines. For instance, SVMs use a shallow linear pattern separation model with one or zero feature transformation layer when the kernel trick is used or otherwise (notable exceptions are the recent kernel methods that have been inspired by and integrated with deep learning).

Shallow architectures have been shown effective in solving many simple or well-constrained problems, but their limited modeling and representational power can cause difficulties when dealing with more complicated real-world applications involving natural signals such as human speech, natural sound and language, and natural image and visual scenes.

Neural networks are one of the most beautiful programming paradigms ever invented. In the conventional approach to programming, we tell the computer what to do, breaking big problems up into many small, precisely defined tasks that the computer can easily perform. By contrast, in a neural network we don't tell the computer how to solve our problem. Instead, it learns from observational data, figuring out its own solution to the problem at hand [1].

Automatically learning from data sounds promising. However, until 2006 we didn't know how to train neural networks to surpass more traditional approaches, except for a few specialized problems. What changed in 2006 was the discovery of techniques for learning in so-called deep neural networks. These techniques are now known as deep learning. They've been developed further, and today deep neural networks and deep learning achieve outstanding performance on many important problems in speech recognition, computer vision and natural language processing. They're being deployed on a large scale by companies such as Google, Microsoft and Facebook.

Three important reasons for the popularity of deep learning today are the drastically increased chip processing abilities (e.g., general-purpose graphical processing units or GPGPUs), the significantly increased size of data used for training, and the recent advances in machine learning and signal/information processing research.

This report studies the deep learning techniques aimed at speech processing, specially focusing on its potential as a method of speech representation for spoken language recognition. We organize the report into three sections. We devote Section 2 to an in-depth look at deep neural networks (DNN). We highlight techniques that are proven to work well in building real systems and explain how and why these techniques work from both theoretical and practical perspectives. We also discuss advanced DNN initialization techniques, including the generative pre-training and fine-tuning, we focus on the restricted Boltzmann machine (RBM).

Section 3 introduces the hybrid and tandem systems for speech recognition and their adaptations to face the spoken language recognition problem.

Section 4 discuses the use of deep learning methods for discovering features in speech signals useful for spoken language identification, and some practical cues result from our experimentation. At the end conclusions are presented in Section 5.

2 Deep Neural Networks

Biologically, neural networks are constructed in a three-dimensional world from microscopic components. These neurons seem capable of nearly unrestricted interconnections. That is not true of any proposed, or existing, man-made network. Integrated circuits, using current technology, are two-dimensional devices with a limited number of layers for interconnection. This physical reality restrains the types, and scope, of artificial neural networks that can be implemented in silicon.

2.1 What are Artificial Neural Networks?

The "art" of using neural networks revolve around the multiple ways these individual neurons can be clustered together. This clustering occurs in the human mind in such a way that information can be processed in a dynamic, interactive, and self-organizing way.

Currently, neural networks are the simple clustering of the primitive artificial neurons. This clustering occurs by creating layers which are then connected one to another. How to connect these layers is the other part of the "art" of engineering networks to resolve real world problems.

Basically, all artificial neural networks have a similar structure or topology as shown in Fig. 2. Although there are useful networks which contain only one layer, or even one element, most applications require networks that contain at least the three normal types of layers: input, hidden, and output. The layer of input neurons receive the data either from input files or directly from electronic sensors in real-time applications. The output layer sends information directly to the outside world, to a secondary computer

process, or to other devices such as a mechanical control system. Between these two layers can be many hidden layers. These internal layers contain many of the neurons in various interconnected structures. The inputs and outputs of each of these hidden neurons simply go to other neurons.

But a neural network (NN) is more than a bunch of neurons. Some early researchers tried to simply connect neurons in a random manner, without much success. Now, it is known that even the brains of snails are structured devices. One of the easiest ways to design a structure is to create layers of elements. It is the grouping of these neurons into layers, the connections between these layers, and the summation and transfer functions that comprises a functioning neural network. The general terms used to describe these characteristics are common to all networks.

2.2 The Deep Neural Network Architecture

There are a large number of different types of networks, but they all are characterised by the following components: a set of nodes and connections between nodes. The nodes can be seen as computational units.

A DNN is a feed-forward [1] artificial neural network (FFNN) that has more than one layer of hidden units between its inputs and its outputs. The basic foundational unit of a NN is the neuron, which is actually conceptually quite simple. As a first look at artificial neural networks could be convenient to understand a type of artificial neuron called perceptron.

A perceptron is a neuron that takes several binary inputs $\{x_1, x_2 ...\}$, and produces a single binary output (see Fig. 1).

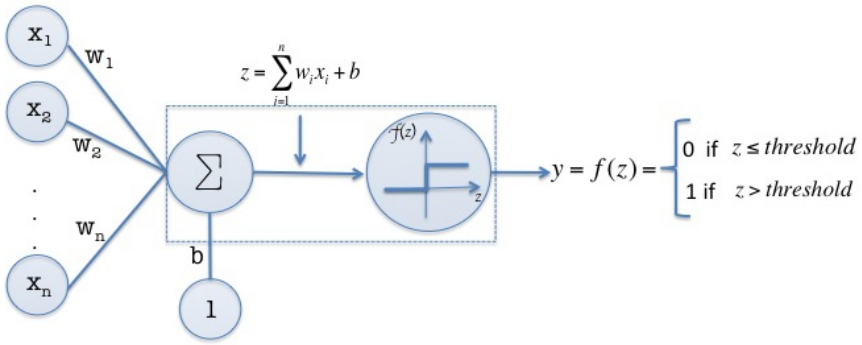

Fig. 1. Perceptron model.

The output is computed using a simple rule which introduces the weights $\{w_1, w_2 ...\}$ as real numbers, expressing the importance of the respective inputs to the output. Basically a perceptron is a device that makes decisions by weighting up evidence:

$$z = \sum_{i=1}^{n} w_i x_i + b. \tag{1}$$

In eq. 1, b is a value added to the sum calculated at each neuron, and it is known as bias. The negative of a bias is sometimes called a threshold. This term increases the capacity of the network to solve problems

[1] The term "feed-forward" indicates that the network has links that extend in only one direction. Except during training, there are no backward links in a feedforward network; all links proceed from input nodes toward output nodes.

and in practice biases are treated in exactly the same manner as other weights, with all biases simply being weights associated to a neuron whose activation is always 1, that takes the form:

$$z = \sum_{i=0}^{n} w_i x_i. \tag{2}$$

The neuron computes some function $f(z)$ on these weighted inputs. But instead of being just 0 or 1, like the perceptron does, these inputs could also take any values between them. This is defined by the transfer function also known as activation function, a list of common activation functions and their names can be found in Appendix 1.

For the sake of notation simplicity, we denote the input layer as layer 0 and the output layer as layer L for an $L+1$-layer DNN. In the first L layers:

$$\mathbf{v}^\ell = f(\mathbf{z}^\ell) = f(\mathbf{W}^\ell \mathbf{v}^{\ell-1} + \mathbf{b}^\ell) \qquad \text{for } 0 < \ell < L, \tag{3}$$

where $\mathbf{z}^\ell = \mathbf{W}^\ell \mathbf{v}^{\ell-1} + \mathbf{b}^\ell \in \mathbb{R}^{N_\ell \times 1}$, $\mathbf{v}^\ell \in \mathbb{R}^{N_\ell \times 1}$, $\mathbf{W}^\ell \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$, $\mathbf{b}^\ell \in \mathbb{R}^{N_\ell \times 1}$, $N_\ell \in \mathbb{R}$ are respectively: the excitation vector, the activation vector, the weight matrix, the bias vector and the number of neurons at layer ℓ.

$\mathbf{v}^0 = \mathbf{o} \in \mathbb{R}^{N_0 \times 1}$ is the observation (or feature) vector, $N_0 = D$ is the feature dimension, and $f(\cdot) : \mathbb{R}^{N_\ell \times 1} \to \mathbb{R}^{N_\ell \times 1}$ is the activation function applied to the excitation vector element.

The output layer needs to be chosen based on the tasks in hand. For the regression tasks, a linear layer is typically used to generate the output vector. For the multi-class classification tasks each output neuron represents a class $i \in 1, ..., C$, where $C = N_L$ is the number of classes. The value of the ith output neuron v_i^L represents the probability $P_{dnn}(i|\mathbf{o})$ that the observation vector $\mathbf{o}$ belongs to class i.

To serve as a valid multinomial probability distribution, the output vector $\mathbf{v}^L$ should satisfy the requirements $v_i^L \geq 0$ and $\sum_{i=1}^{C} v_i^L = 1$. This can be done by normalizing the excitation with a softmax function

$$v_i^L = P_{dnn}(i|\mathbf{o}) = \text{softmax}_i\left(\mathbf{z}^L\right) = \frac{e^{z_i^L}}{\sum_{j=1}^{C} e^{z_j^L}}, \tag{4}$$

where z_i^L is the ith element of the excitation vector $\mathbf{z}^L$ (also see Appendix 1).

However in a NN, whatever function the neuron uses, the value it computes is transmitted to other neurons as its input. Fig. 2 visualises such structure with a simple example of a neural net.

Given an observation vector $\mathbf{o}$, the output of the DNN specified by the model parameters $\{\mathbf{W}, \mathbf{b}\} = \{\mathbf{W}^\ell, \mathbf{b}^\ell | 0 < \ell \leq L\}$ can be calculated by computing the activation vectors with eq. 3 layer by layer from layer 1 to layer $L-1$ and with eq. 4 for the classification tasks to calculate the output of the DNN. This process is often called forward computation.

2.2.1 The Backpropagation Algorithm

The backpropagation algorithm (BP) looks for the minimum of the error function in the weight space using the method of gradient descent. The combination of weights which minimizes the error function is considered to be a solution of the learning problem. Since this method requires computation of the gradient of the error function at each iteration step, we must guarantee the continuity and differentiability of the error function. Obviously we have to use a kind of activation function other than the step function used in perceptrons, because the composite function produced by interconnected perceptrons is discontinuous, and therefore the error function too, that is one of the reasons why DNNs are not MLPs.

The model parameters $\mathbf{W}, \mathbf{b}$ in a DNN, however, are unknown and need to be estimated from training samples $\mathbb{S} = \{(\mathbf{o}^m, \mathbf{t}^m) | 0 < m \leq M\}$ for each task, where M is the number of training samples, $\mathbf{o}^m$ is the mth

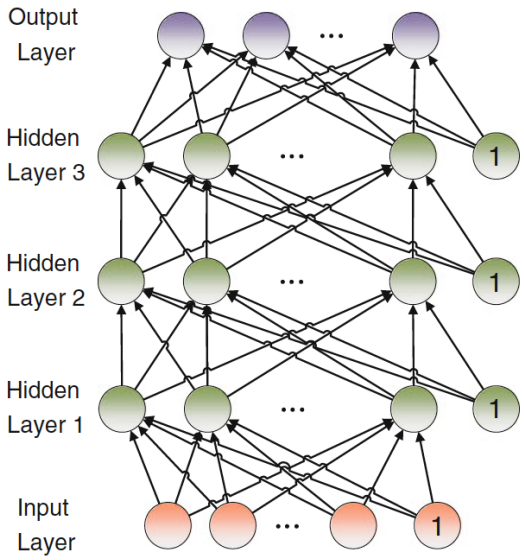

Fig. 2. An example deep neural network with an input layer, three hidden layers, and an output layer (Figure taken from [2]).

observation vector and $\mathbf{t}^m$ is the corresponding desired output vector or target. This process is often called the training process or the parameter estimation process, which can be specified by a training criterion and a learning algorithm.

Consider a FFNN with N_0 input and N_L output units. It can consist of any number of hidden units and can exhibit any desired feed-forward connection pattern. We are also given a training set $\mathbb{S}$ defined above consisting in M ordered pairs of D and C-dimensional vectors, which are called the input and output vectors. Let the activation functions at each node of the network be continuous and differentiable. The weights are real numbers selected at random. When the input pattern $\mathbf{o}^m$ from the training set is presented to this network, it produces an output $\mathbf{v}^m$ different in general from the target $\mathbf{t}^m$. What we want is to make $\mathbf{v}^m$ and $\mathbf{t}^m$ identical for $m = 1,...,M$ by using a learning algorithm. More precisely, we want to minimize the error function of the network, defined as:

$$E = \frac{1}{2}\sum_{m=1}^{M} \|\mathbf{v}^m - \mathbf{t}^m\|^2. \tag{5}$$

After minimizing this function for the training set, new unknown input patterns are presented to the network and we expect it to interpolate. The network must recognize whether a new input vector is similar to learned patterns and produce a similar output.

The BP algorithm is used to find a local minimum of the error function. The network is initialized with randomly chosen weights. The gradient of the error function is computed and used to correct the initial weights. Our task is to compute this gradient recursively.

Every one of the j output units of the network is connected to a node which evaluates the function $\frac{1}{2}\sum_{m=1}^{M}(v_j^m - t_j^m)^2$, where v_j^m and t_j^m denote the jth component of the output vector $\mathbf{v}^m$ and the target $\mathbf{t}^m$. The outputs of the additional C nodes are collected at a node which adds them up and gives the sum E^m as its output. The same network extension has to be built for each pattern $\mathbf{t}^m$. A computing unit collects all quadratic errors and outputs their sum $E^1 + ... + E^M$. The output of this extend network is the error function E.

We now have a network capable of calculating the total error for a given training set. The weights in the network are the only parameters that can be modified to make the quadratic error E as low as possible.

Because E is calculated by the extended network exclusively through composition of the node functions, it is a continuous and differentiable function of the K weights $\{w_1, w_2, ..., w_K\}$ in the network. We can thus minimize E by using an iterative process of gradient descent, for which we need to calculate the gradient

$$\nabla E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, ..., \frac{\partial E}{\partial w_K}). \tag{6}$$

Each weight is updated using the increment

$$\Delta w_k = -\gamma \frac{\partial E}{\partial w_k} \text{ for } k = 1, ..., K, \tag{7}$$

where γ represents a learning constant, a proportionality parameter which defines the step length of each iteration in the negative gradient direction. With this extension of the original network the whole learning problem now reduces to the question of calculating the gradient of a network function with respect to its weights. Once we have a method to compute this gradient, we can adjust the network weights iteratively. In this way we expect to find a minimum of the error function, where $\nabla E = 0$.

Given the training criterion, the model parameters $\{\mathbf{W}, \mathbf{b}\}$ can be learned with the BP algorithm.

2.3 Practical Considerations

The basic BP algorithm described in subsection 2.2.1 is theoretically simple. However, learning a model efficiently and effectively requires taking in consideration many practical issues [3].

2.3.1 Data Pre-processing

Data preprocessing plays a very important role in many deep learning algorithms. In practice, many methods work better after the data has been normalized and whitened. However, the exact parameters for data preprocessing are usually not immediately apparent unless one has much experience working with the algorithms.

A standard first step to data preprocessing is data normalization. While there are a few possible approaches, this step is usually clear depending on the data. The common methods for feature normalization are:

- Simple Rescaling
- Per-example mean subtraction
- Feature Standardization (zero-mean and unit variance for each feature across the dataset)

In simple rescaling, our goal is to rescale the data along each data dimension (possibly independently) so that the final data vectors lie in the range [0,1] or [1,1] (depending on your dataset). This is useful for later processing as many default parameters treat the data as if it has been scaled to a reasonable range.

If your data is stationary (the statistics for each data dimension follow the same distribution) or the mean of each sample reflects a variation that is irrelevant to the problem in hand, then you might want to consider subtracting the mean-value for each example, to reduce the variability in the final feature fed to the DNN.

Feature standardization refers to (independently) setting each dimension of the data to have zero-mean and unit-variance. This is the most common method for normalization and is widely used. In practice, one achieves this by first computing the mean of each dimension (across the dataset) and subtracts this from each dimension [4]. Next, each dimension is divided by its standard deviation. The goal of feature

standardization is to scale the data along each dimension using a global transformation so that the final data vectors lie in a similar range. For example, when working with audio data, it is common to use mel-frequency cepstral coefficients (MFCCs) as the data representation. However, the first component (representing the energy) of the MFCC features often overshadow the other components. Thus, one method to restore balance to the components is to standardize the values in each component independently. This standardized features allow, in DNN training, to use the same learning rate across all weight dimensions and still get a good model.

2.3.2 Model Initialization

There are many heuristic tricks in initializing the DNN model. Most of these tricks are based on two considerations:

1. The weights should be initialized so that each neuron operates in the linear range of the sigmoid function at the start of the learning.
2. It is important to initialize the parameters randomly.

If weights were all very large, many neurons would saturate (close to zero or one) and the gradients would be very small. When the neurons operate in the linear range, instead, the gradients are large enough (close to the maximum value of 0.25) that learning can proceed effectively.

On the other hand, random initialization serves the purpose of symmetry breaking.

However, for deep architectures model initialization must be take very carefully. Thanks to Hinton's pre-training procedure [5], the parameters of the deep neural network with a lot of hidden layers and a huge output layer can still be learned in a very reliable way (see also Section 2.4).

2.3.3 Weight Decay

As in many machine learning algorithms, overfitting can be a problem especially since the number of parameters in a DNN is huge compared to many other machine learning techniques.

The learning data contains information about the regularities in the mapping from input to output. But it also contains sampling error, that means that there will be accidental regularities just because of the particular training cases that were chosen. When we fit the model, it can not tell which regularities are real and which are caused by sampling errors (spurious), so the model fits both kinds of regularities. If the model is very flexible it can model the sampling error really well and it will generalize badly.

Increasing the complexity of the model does not ensure generalization power, even tough it reduces the empirical losses defined on the training set (see Fig.9).

There are many ways to prevent overfitting, one of them is to use a model with the "right" capacity, in a way that the model will be able to fit the true regularities, but it wont have enough capacity to fit the spurious caused by the sampling error. A very common way to control the capacity of the neural networks is to limit the size of the weights by penalizing them.

The standard way to limit the size of the weight is to use an $L2$ weight penalty, which means that we penalize the square value of the weight. This is called weight decay in the neural network literature because the derivative of that penalty acts like a force pulling the weights to zero.

If we look at the eq. 8 the cost that we are optimizing (C) is the error that we are trying to reduce plus a term where λ is the weight cost that determines how strong the penalty is:

$$C = E + \frac{\lambda}{2}\sum_i w_i^2. \tag{8}$$

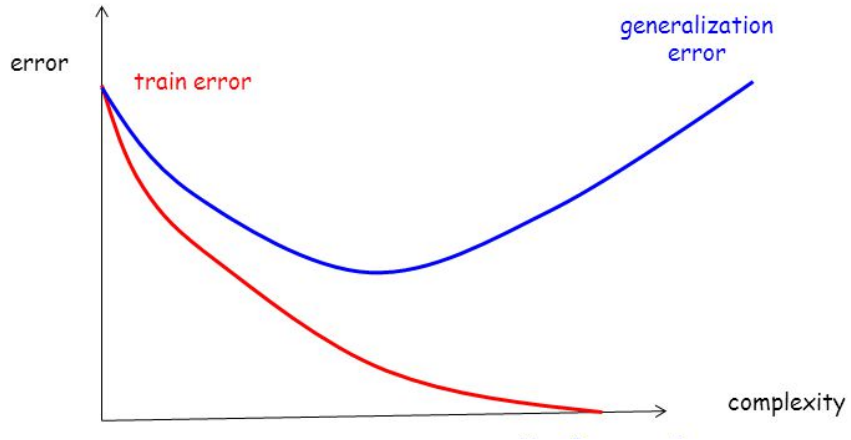

Fig. 3. Generalization problem: over-fitting and under-fitting.

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i. \tag{9}$$

$$\frac{\partial C}{\partial w_i} = 0 \Rightarrow w_i = -\frac{1}{\lambda}\frac{\partial E}{\partial w_i}. \tag{10}$$

The derivative will be zero when the magnitude of the weight is 1 over λ times the magnitude of the derivative. So the only way you can have big weights when you are at the minimum of the cost function is if they also have big error derivatives, which guarantee that you wont have a lot of useless large weights.

The effect of $L2$ penalty on the weights is to prevent the network from using weights that it doesn' t need. This often improve generalization because it stops network from fitting the sampling error. It also makes a smoother model in which the output changes more slowly as the input changes.

2.3.4 Dropout

Another popular approach to control overfitting is dropout [6]. The term dropout refers to dropping out units (hidden and visible) in a neural network. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections, as shown in Figure 4.

The choice of which units to drop is random. In the simplest case, each unit is retained with a fixed probability **p** independent of other units, where **p** can be chosen using a validation set or can simply be set at 0.5, which seems to be close to optimal for a wide range of networks and tasks.

Dropout is a technique that prevents overfitting and provides a way of approximately combining exponentially many different neural network architectures efficiently.

Applying dropout to a neural network amounts to sampling a "thinned" network from it. The thinned network consists of all the units that survived dropout (Figure 4(b)). A neural net with n units, can be seen as a collection of 2^n possible thinned neural networks. These networks all share weights so that the total number of parameters is still $O(n^2)$, or less. For each presentation of each training case, a new thinned network is sampled and trained.

So training a neural network with dropout can be seen as training a collection of 2^n thinned networks with extensive weight sharing, where each thinned network gets trained very rarely. At test time, it is not

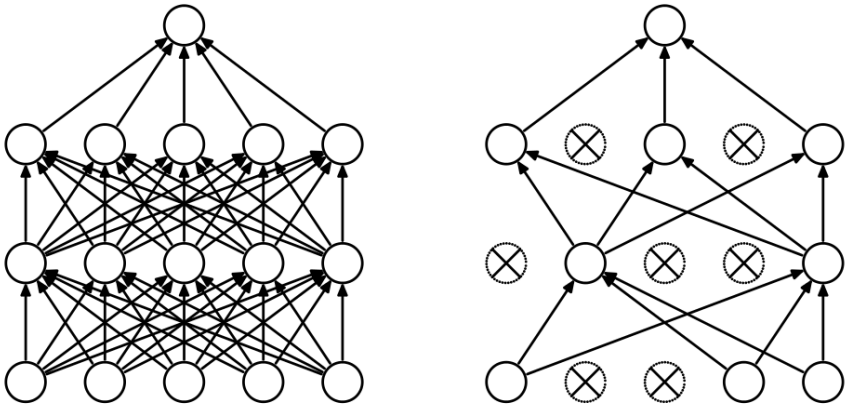

Fig. 4. (a)Standard neural net. (b)After applying dropout.

feasible to explicitly average the predictions from exponentially many thinned models. However, a very simple approximate averaging method works well in practice.

It is known that training a network with dropout and using an averaging method at test time leads to significantly lower generalization error on a wide variety of classification problems compared to training with other regularization methods.

Each hidden unit in a neural network trained with dropout must learn to work with a randomly chosen sample of other units. This should make each hidden unit more robust and drive it towards creating useful features on its own without relying on other hidden units to correct its mistakes. Complex co-adaptations can be trained to work well on a training set, but on novel test data they are far more likely to fail than multiple simpler co-adaptations that achieve the same thing.

2.3.5 Batch Size Selection

It is possible to update the weights after estimating the gradient on a single training case, this is known as the stochastic gradient descent technique (SGD) [7].

The SGD algorithm, is difficult to parallelize even on the same computer. In addition, it cannot fully converge to the local minimum due to the noisy estimation of the gradient. Instead, it will fluctuate around the minimum. The size of the fluctuation depends on the learning rate and the amplitude of the gradient estimation variance. Although this fluctuation can sometimes reduce overfitting, it is not desirable in many cases.

To avoid this, the training set is divided into small mini-batches of 10 to 100 cases. The total gradient computed on a mini-batch is divided by the size of the mini-batch, so when talking about learning rates we will assume that they multiply the average per-case gradient computed on a mini-batch. It is a serious mistake to make the mini-batches too large when using SGD. Increasing the mini-batch size by a factor of N leads to a more reliable gradient estimate but it does not increase the maximum stable learning rate by a factor of N, so the net effect is that the weight updates are smaller per gradient evaluation.

2.3.6 Momentum

A simple modification of BP called BP with momentum was introduced to minimize the error function using gradient descent. The gradient of the error function is computed for each new combination of weights, but instead of just following the negative gradient direction a weighted average of the current gradient and the previous correction direction is computed at each step. Theoretically, this approach should provide the

search process with a kind of inertia and could help to avoid excessive oscillations in narrow valleys of the error function.

As explained before, in standard BP the input-output patterns are fed into the network and the error function E is determined at the output. When using BP with momentum in a network with K different weights $\{w_1, w_2, ..., w_K\}$, the i-th correction for weight w_k is given by

$$\Delta w_k(i) = -\gamma \frac{\partial E}{\partial w_k} + \alpha \Delta w_k(i-1), \tag{11}$$

where γ and α are the learning and momentum rate respectively. Normally, we are interested in accelerating convergence to a minimum of the error function, and this can be done by increasing the learning rate up to an optimal value.

Several fast learning algorithms for neural networks work by trying to find the best value of which still guarantees convergence. Introduction of the momentum allows the attenuation of oscillations in the iteration process. The adjustment of both learning parameters to obtain the best possible convergence is normally done by trial and error. Since the optimal parameters are highly dependent on the learning task, no general strategy has been developed to deal with this problem.

2.3.7 Sample Randomization

One important point regarding SGD is the order in which we present the data to the algorithm. If the data is given in some meaningful order, this can bias the gradient and lead to poor convergence. If successive samples are not randomly drawn from the training set (e.g., all belong to the same speaker), the model parameters will likely to move along the similar direction for too long. Generally a good method to avoid this is to randomly shuffle the data prior to each epoch of training.

Computing an unbiased estimate of the expected gradient from a set of samples requires that those samples be independent. We also wish for two subsequent gradient estimates to be independent from each other, so two subsequent minibatches of examples should also be independent from each other. Many datasets are most naturally arranged in a way where successive examples are highly correlated. For example, in cases where the order of the dataset holds some significance, it is necessary to shuffle the examples before selecting minibatches.

If the whole training set can be loaded into the memory, sample randomization can be easily done by permuting an index array. Samples can then be drawn one by one according to the permuted index array.

For very large datasets, containing billions of examples in a data center, it can be impractical to sample examples truly uniformly at random every time we want to construct a minibatch. Fortunately, in practice it is usually sufficient to shuffle the order of the dataset once and then store it in shuffled fashion.

This will impose a fixed set of possible minibatches of consecutive examples that all models trained thereafter will use, and each individual model will be forced to reuse this ordering every time it passes through the training data. However, this deviation from true random selection does not seem to have a significant detrimental effect. Failing to ever shuffle the examples in any way can seriously reduce the effectiveness of the algorithm.

2.4 Advanced Model Initialization Techniques

Now we introduce some advanced deep neural network (DNN) model initialization or pretraining techniques. These techniques have played important roles in the early days of deep learning research and continue to be useful under many conditions. We focus our presentation of pretraining DNNs on the restricted Boltzmann machine (RBM), which is an interesting generative model.

In [5], it has been proposed to use a pre-training algorithm based on restricted Boltzmann machine (RBM) to initialize DNN. RBM is a two-layer generative model based on an energy function assigned to every configuration of visible and hidden state vectors. An efficient unsupervised algorithm based on one-step contrastive divergence (CD) as in [5] is available to learn all connection weights between two layers in RBM. This pre-training algorithm has been found to be also effective in training DNN, where the learned weights of RBM can be directly used to initialize a two-layer feed-forward neural network with sigmoid hidden units. Once we have trained an RBM from training data, we can use its hidden activation probabilities as training data to train another layer of RBM, which in turn is used to initialize next layer in DNN.

In such a way, we will be able to initialize all weights in all layers of DNN using the above RBM connection weights. This is called pre-training. After that, we add a randomly initialized soft max output layer and use the standard BP algorithm to fine-tune all parameters in DNN. See [5,4] for more details on RBM-based pre-training.

2.4.1 RBMs

The restricted Boltzmann machine (RBM) [8] as its name indicates, it is a variant of the Boltzmann machine. It is essentially an undirected graphical model constructed of a layer of stochastic visible neurons and a layer of stochastic hidden neurons.

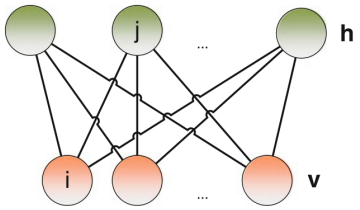

Fig. 5. An example of restricted Boltzmann machines (Figure taken from [2]).

The visible and hidden neurons form a bipartite graph with no visible-visible or hidden-hidden connections as shown in Fig.5. The hidden neurons usually take binary values and follow Bernoulli distributions. The visible neurons may take binary or real values depending on the input types.

An RBM assigns an energy to every configuration of visible vector $\mathbf{v}$ and hidden vector $\mathbf{h}$. For the Bernoulli-Bernoulli RBM, in which $\mathbf{v} \in \{0,1\}^{N_v \times 1}$ and $\mathbf{h} \in \{0,1\}^{N_h \times 1}$, the energy is

$$E(\mathbf{v},\mathbf{h}) = -\mathbf{a}^T\mathbf{v} - \mathbf{b}^T\mathbf{h} - \mathbf{h}^T\mathbf{W}\mathbf{v}, \tag{12}$$

where N_v and N_h are the number of visible and hidden neurons, respectively, $\mathbf{W} \in \mathbb{R}^{N_h \times 1}$ is the weight matrix connecting visible and hidden neurons, and $\mathbf{a} \in \mathbb{R}^{N_h \times 1}$ and $\mathbf{b} \in \mathbb{R}^{N_h \times 1}$ respectively are, the visible and the hidden layer bias vectors.

If the visible neurons take real values, $\mathbf{v} \in \mathbb{R}^{N_v \times 1}$, the RBM, which is often called Gaussian-Bernoulli RBM, assigns the energy

$$E(\mathbf{v},\mathbf{h}) = \frac{1}{2}(\mathbf{v}-\mathbf{a})^T(\mathbf{v}-\mathbf{a}) - \mathbf{b}^T\mathbf{h} - \mathbf{h}^T\mathbf{W}\mathbf{v} \tag{13}$$

to each configuration $(\mathbf{v},\mathbf{h})$.

Each configuration is also associated with a probability

$$P(\mathbf{v},\mathbf{h}) = \frac{1}{Z}e^{-E(\mathbf{v},\mathbf{h})}, \tag{14}$$

that is defined upon the energy, where $Z = \sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}$ is the normalization factor also known as the partition function.

In the RBM the posterior probabilities $P(\mathbf{v}|\mathbf{h})$ and $P(\mathbf{h}|\mathbf{v})$ can be efficiently calculated thanks to the lack of direct connections within visible and hidden layers:

$$P(\mathbf{h} = 1|\mathbf{v}) = \sigma(\mathbf{W}\mathbf{v} + \mathbf{b}), \tag{15}$$

where $\sigma(x) = (1 + e^{-x})^{-1}$ is the element-wise logistic sigmoid function. For the binary visible neuron case, a completely symmetric derivation lets us obtain

$$P(\mathbf{v} = 1|\mathbf{h}) = \sigma(\mathbf{W}\mathbf{h} + \mathbf{a}). \tag{16}$$

For the Gaussian visible neurons, the conditional probability

$$P(\mathbf{v} = 1|\mathbf{h}) = \mathcal{N}(\mathbf{v}; \mathbf{W}\mathbf{h} + \mathbf{a}, I), \tag{17}$$

where I is the appropriate identity covariance matrix.

2.4.2 Contrastive Divergence

Unlike DNN, in an RBM the gradient of the log likelihood of the data is not feasible to compute exactly. For the visible-hidden weight updates we have:

$$\Delta w_{ij} = \gamma[\langle v_i h_j \rangle_{data} - \langle v_j h_i \rangle_{model}]. \tag{18}$$

The first expectation, $\langle v_i h_i \rangle_{data}$, is the frequency with which the visible neuron v_i and the hidden neuron h_j fire together in the training set and $\langle v_j h_j \rangle_{model}$ is that same expectation under the distribution defined by the model. Unfortunately, the term $\langle \cdot \rangle_{model}$ takes exponential time to compute exactly when the hidden values are unknown, so we are forced to use approximated methods.

The most widely used efficient approximated learning algorithm for the RBM training is contrastive divergence (CD) as described in [9]. The one-step contrastive divergence approximation for the gradient with regard to the visible-hidden weights is

$$\begin{aligned} \Delta w_{ij} &= \gamma[\langle v_i h_j \rangle_{data} - \langle v_j h_i \rangle_{\infty}], \\ &\approx \gamma[\langle v_i h_j \rangle_{data} - \langle v_j h_i \rangle_{1}], \end{aligned} \tag{19}$$

where $\langle \cdot \rangle_{\infty}$ and $\langle \cdot \rangle_1$ denote the expectation computed with samples generated by running the Gibbs sampler infinite steps and one step, respectively. Fig.6 illustrates the sampling process and the CD algorithm.

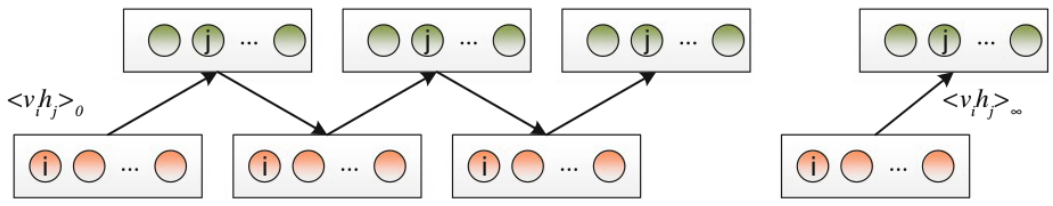

Fig. 6. Illustration of the contrastive divergence algorithm (Figure taken from [2]).

At the first step, the Gibbs sampler is initialized at a data sample. It then generates a hidden sample from the visible sample based on the posterior probability $P(\mathbf{h}|\mathbf{v})$ defined by eq.15. This hidden sample is further used to generate a visible sample based on the posterior probability $P(\mathbf{v}|\mathbf{h})$ defined by eq.16 for the Bernoulli-Bernoulli RBMs or eq.17 for the Gaussian-Bernoulli RBMs. This process continues and may run many steps. However, running Gibbs sampler many steps is not efficient. Instead, we can replace it with a very rough estimate by running the Gibbs sampler just one step

$$\langle v_i h_j \rangle_{model} \approx \langle v_i h_j \rangle_1. \tag{20}$$

Similar to the DNN training, effective RBM training also requires many practical considerations. Many of the discussions in Section 2.3 can be applied to the RBM training. A comprehensive practical guide in training RBMs can be found in [10].

3 DNNs Applications over Speech Processing Systems

In speech related tasks, there are mainly two different ways to incorporate deep learning techniques [11]. In the first configuration, which is called a DNN-Hidden Markov Model (HMM) hybrid or simply hybrid, a DNN is used to compute the posterior probabilities of context-dependent HMM states based on observed feature vectors [12], a Viterbi decoding is then performed with these posteriors.

As an alternative to GMM in large vocabulary continuous speech recognition (LVCSR), this hybrid approach recently became popular since it had been discovered that the DNN based estimation of posterior probabilities of thousands of tied triphone states is feasible. The introduction of deep structures further increased the effectiveness of the ANN as acoustic model [13]. The DNN-HMM hybrid system takes advantage of DNN's strong representation learning power and HMM's sequential modeling ability, and outperforms conventional GMM-HMM systems significantly on many LVCSR tasks.

The second configuration, which is called tandem, uses the DNN to perform a nonlinear discriminative feature transformation, which yields DNN features [14]. These features are merged with a set of standard ones, such as MFCCs or PLP coefficients, to form a new set, which are collectively called tandem features, and then input into a GMM acoustic model.

In the more specific bottle-neck tandem concept proposed by [15], a DNN consisting of at least 5 layers with a narrow one in the middle is trained, whereas the linear output of the middle (bottle-neck) layer is taken as output instead of the posteriors. The biggest advantage of the tandem approach is that all techniques developed for GMMs in the previous decades remain applicable, e.g. speaker adaptive and discriminative training. As we said before the tandem approach uses the posteriors as features in a GMM or i-vector based system.

3.1 DNN-HMM Hybrid Systems

The DNNs we described in Section 2 cannot be directly used to model speech signals since speech signals are time series signals while DNNs require fixed size inputs. To exploit the strong classification ability of DNNs in speech recognition, we need to find a way to handle the variable length problem in speech signals. The combination of artificial neural networks (ANNs) and HMMs as an alternative paradigm for ASR started between the end of the 1980s and the beginning of the 1990s. A variety of different architectures and training algorithms were proposed at that time [16].

This line of research was resurrected recently after the strong representation learning power of DNNs became well known. One of the approaches that have been proven to work well is to combine DNNs with

HMMs in a framework called DNN-HMM hybrid system as illustrated in Fig. 7. In this framework, the dynamics of the speech signal is modeled with HMMs and the observation probabilities are estimated through DNNs. Each output neuron of the DNN is trained to estimate the posteriors probability of continuous density HMM state given the acoustic observation.

These kinds of hybrid models were proposed and seen as a promising technique for LVCSR in the early to mid-1990s and were referred as ANN-HMM hybrid models in the literature [17].

These earlier hybrid models have some important limitations. For example, ANNs with more than two hidden layers were rarely exploited due to the computational power limitations, and the context-dependent model does not take advantage of the numerous effective techniques developed for GMM-HMMs.

A recent advancement indicates that significant recognition accuracy improvement can be achieved if we replace the traditional shallow neural networks with deeper (optionally pretrained) ones, and use senones (tied triphone states) instead of monophone states as the output units of the neural network [4]. This improved ANN-HMM hybrid model is referred as CD-DNN-HMM.

CD-DNN-HMMs outperform GMM-HMMs in many LVCSR tasks. The components or procedures that have contributed most to the performance improvement are:

- using deep neural networks with sufficient depth,
- using a long window of frames, and
- model senones directly.

The CD-DNN-HMM system replaces the GMM with likelihoods derived from the DNN posteriors, while leaving everything else the same.

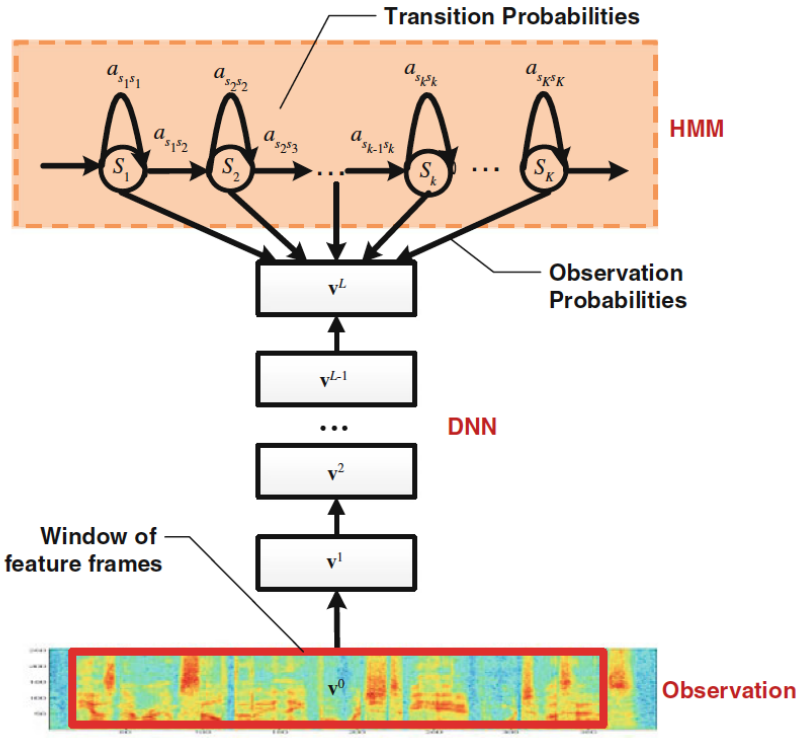

Fig. 7. Architecture of the DNN-HMM hybrid system. The HMM models the sequential property of the speech signal, and the DNN models the scaled observation likelihood of all the senones (tied triphone states). The same DNN is replicated over different points in time (figure taken from [2]).

3.2 Tandem

The idea of treating the hidden and output layers as better features was first proposed for the shallow multilayer perceptron (MLP) under the name of Tandem approach [14]. The Tandem approach augments the input to a GMM-HMM system with features derived from the suitably transformed output of one or more neural networks.

Since the size of the output layer is the same as that of the training target, Tandem features are typically trained to produce distributions over monophone targets to control the dimension of the augmented feature. Alternatively, Grezl et al. [15,18] proposed to use features derived from a bottleneck hidden layer, which has smaller size than that of other layers, instead of using the neural network outputs directly. Since the hidden layer size can be chosen independent of the output layer size, this provides flexibility in choosing the training targets and the size of the augmented feature. The bottleneck layer creates a constriction in the network and forces the information pertinent to classification into a low-dimensional representation.

More recently, DNNs are in place of shallow MLPs to extract more robust features. These DNNs are often trained to classify senones instead of monophone labels. For this reason, a hidden layer, instead of the output layer, is often used as the feature to the GMM system.

Since the hidden layer feature will be modeled using GMMs, we should use excitations (i.e., before the nonlinear activation function is applied) instead of activations, especially if the sigmoid nonlinearity is used since sigmoid will generate features that are bounded within [0, 1] and concentrated in the two extreme values 0 and 1. Further more, even if we use bottleneck layers the dimension of the bottleneck feature (BNF) may still be large and somewhat correlated between dimensions. For this reason it is often still useful to apply PCA or HLDA before it is used in the GMM-HMM system. Note that since the Tandem (or bottleneck) features are trained independent of the GMM-HMM system, it is difficult to know which layer provides the best feature and whether adding more layers would perform better.

3.3 DNN-HMM Hybrid System vs. Tandem System with DNN-Derived Features

The main difference between the DNN-HMM hybrid system and the GMM-HMM system that uses Tandem or bottleneck features is the classifier used. In the Tandem / bottleneck system, GMM is used in place of log-linear model (the softmax layer in DNNs) used in the DNN-HMM system. On the other hand, since the hidden layers are trained jointly with the log-linear classifier, they fit better to the log-linear model (i.e., in the DNN-HMM hybrid system) than the Tandem/bottlenck system. The net effect of these two factors cancels out and the two types of systems perform almost equally well when averaged over different tasks. However, the integrated CD-DNN-HMM system is simpler conceptually and in practice [2].

The main benefit of using DNN-derived features in GMM-HMM systems is the possibility of taking advantage of existing tools already available to train and to adapt GMM-HMM systems. It also allows DNNs used to derive features being trained with a subset of the training data and the GMM-HMM system that uses the DNN-derived features being trained with the full training set. Tandem approach is the one selected by us to conduct our experiments and to set our base line system for language identification (LID).

4 Representation Learning for LID

Language identification (LID) is the task of determining the identity of the spoken language present within a speech utterance. LID is a key pre-processing technique for multi-lingual speech processing systems,

such as audio and video information retrieval, automatic machine translation, diarization, multi-lingual speech recognition and intelligent surveillance among others.

A major problem in LID is how to design a language specific and effective representation for speech utterances. It is challenging due to large variations introduced by different speech content, speakers, channels and background noises. Over the past few decades, intensive research efforts have studied the effectiveness of different representations from various research domains, such as phonotactic and acoustic information [19,20,21], lexical knowledge [22], prosodic information, articulatory parameters [23], and universal attributes [24].

Among existing representations, it has been shown that appropriate incorporation of extra language-related cues may help to improve the effectiveness of representation, however we will mainly focus on the phonotactic and acoustic representations, which are considered to be the most common ones for LID [25].

The phonotactic representations use the output of a specific phone recogniser (PR), and enjoy the power of robust language modelling techniques for contextual information, e.g. PR with language model (PRLM) and PR with support vector machine [26]. The acoustic representations are derived from a GMM model based on the shifted delta cepstra (SDC) feature, which is a linear expansion of the MFCC with fixed structure as in [27]. Compared with the phonotactic representations, acoustic one may take advantage of lower computational complexity, and do not need extra labelling for PR training. Recently, proposed acoustic modelling methods, such as GMM-SVM, factor analysis and i-Vector -also known as Total Variability (TV)-[28], can achieve comparable or even better performance.

However, performance is still far from satisfactory for highly confusable dialects, short duration utterances, noisy conditions and low resources languages. This may be because language information is latent and largely dependent on the statistical distribution of extracted features. For short-duration utterances and for dialects, existing representations are clearly insufficient. They are also susceptible to variations introduced by different speech content, speakers, channels and background-noise.

Differing from phonotactic approaches, acoustic one aim to capture the difference between languages by modelling spectral feature distributions, as well as being inherently efficient [21]. Acoustic features play an important role in the classical frame-level feature based LID approaches. Currently, most existing systems use SDC coefficients as acoustic features. In general, SDC can be viewed as an extension of the short-term spectral features, such as MFCC and Perceptual Linear Prediction, which can capture temporal information over a longer time window size. SDC enables acoustic approaches to perform as well as phonotactic ones [27]. Despite the success of SDC, several issues remain:

- SDC is a linear feature transformation which may not be optimal;
- Useful language information is often distorted by language-independent nuisance, e.g. channel and speaker variations;
- The parameters used for extracting SDC are a matter of trial and error.

Recently, DNNs have achieved significant performance gains either as acoustic models or for front-end bottleneck feature (BNF) extraction [29,30] in many challenging speech and language recognition tasks. Motivated by the success of DNN, we planed to use BNF as an alternative to SDC to train acoustic models for LID based on i-vector representation as presented in [31], see Fig.8

In [32], it is shown that deep bottleneck features (BNF), the output from a constricted internal layer of a structured Deep Bottleneck Network (DBN), can effectively mine the contextual information embedded in speech frames. By representing each utterance as an i-vector, these LID systems achieved excellent performance in the NIST LRE2009 evaluation.

This is broadly speaking, the baseline chosen by us. Our motivation is that, if the pre-trained DBN can be considered as a bridge from low-level spectral or acoustic features to high-level phonetic features, then

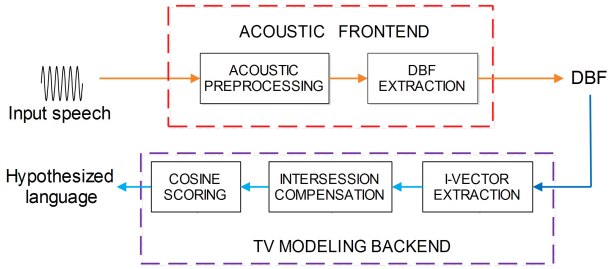

Fig. 8. i-vector representation based on BNF for automatic LID (Figure taken from [31]).

the output from different DBN layers may represent a graded mixture of acoustic and phonetic information. Exploiting this may be advantageous for LID.

Specifically, with a well-trained DBN structure as frontend feature extractor, a lot of different evaluations measuring the effectiveness of representations from the internal bottleneck layer and the topmost layer, could be done. However to reach that initial goal of a well-trained DBN structure, it is not a trivial task.

4.1 Methods

The LID system assumed as baseline, mainly consists of two parts: acoustic frontend (DBN structure) and Total Variability modeling backend (TV).

The DNN training stage was done using the Deep Learning Library (DLL) developed by Baptiste Wicht. DLL[2] is a small library that aims to provide a $C++$ implementation of Restricted Boltzmann Machine (RBM) and Deep Belief Network (DBN). Some available toolboxes for DNN implementation can be found in Appendix 2.

The second part (TV modeling backend) wont be addressed in this report, as it is part of a very well established procedure and it is not a deep learning technique.

4.1.1 DNN Training insights

The DNN training process includes pre-training and fine-tuning phases [30].

During the pre-training phase, a generative Deep Belief Network (DBN) with stacked Restricted Boltzmann Machines (RBM) is trained in an unsupervised way. During the discriminative fine-tuning phase, a randomly initialized softmax layer is added on top of the DBN, and all the parameters are fine-tuned jointly using BP. Generally, the pre-training phase provides a region of the weight space that allows the fine-tuning phase to converge to a better local optimum, and reduce overfitting [33].

Pre-Training Phase

The basic idea of pre-training is to fit a generative DBN model to the input data. The DBN can be trained in a layer-by-layer manner, by treating each pair of layers as a RBM [5], as shown in the left part of Fig.9.

As we discussed before (2.3.1) data pre-processing plays a fundamental role in the success of the DNN training process. In our experiments the feature dimension of each frame was set to 43, consisting of 39-dimensional $MFCC + \Delta MFCC + \Delta\Delta MFCC$ and 4-dimensional pitch features corresponding to the static pitch, 1st and 2nd derivatives and the normalized energy. The frame feature is pre-processed with

[2] The $C++$ implementation of DLL is freely available: https://github.com/wichtounet/ana_template

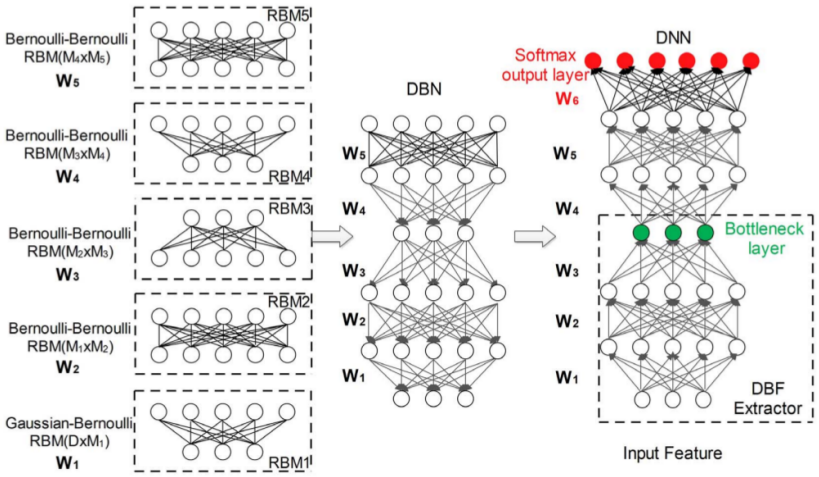

Fig. 9. An illustration of the DNN training and DBF extraction procedure (Figure taken from [31]).

Cepstral Mean Variance Normalization (CMVN). The input feature is constructed in a frame by frame manner. For each fame, the corresponding DNN input is a concatenation of the current frame with the preceding and following $(n-1)/2$ neighbouring frames. For example, if we set $n = 11$, the input comprises 5 neighbouring frames before and after the center frame.

For the pre-training phase, no labeled data is needed, as the RBM training is unsupervised.

For the first RBM, as the inputs were real values, Gaussian units were used. For the rest of the stacked RBM the units were set to stochastic and binary, except for the output layer were Softmax was the activation function 1.

Different configurations were tested playing with the number of layers and units per layer. However an important practical aspect is to always start with a reduced dataset, which necessarily leads you to the use of simpler models. This is a very healthy procedure, even when the tuned model wont likely work over the large scale problem, it will show if the network is able to learn something from the selected input features.

Monitoring the progress of learning is a key aspect in the DNN training process. It is easy to compute the squared error between the data and the reconstructions, so this quantity is often printed out during learning. The reconstruction error on the entire training set should fall rapidly and consistently at the start of learning and then more slowly. Due to the noise in the gradient estimates, the reconstruction error on the individual mini-batches will fluctuate gently after the initial rapid descent. It may also oscillate gently with a period of a few mini-batches when using high momentum.

Although it is convenient, the reconstruction error is actually a very poor measure of the progress of learning [10]. It is not the function that CD_n learning is approximately optimizing, especially for $n >> 1$ [3], and it systematically confounds two different quantities that are changing during the learning. Large increases, however, are a bad sign except when they are temporary and caused by changes in the learning rate, momentum or weight-cost meta-parameters.

In order to design a reduced scale experiment, is very important to pay attention to the representativeness of the training set and to its randomization at feature level, something that wasn't tackled in our approach because of particularities of DLL design, and that probably affected the convergence of the network.

[3] CD_n will be used to denote learning using n full steps of alternating Gibbs sampling (see Section 2.4).

A lot of time and efforts are dedicated to tune the net to achieve the proper convergence. Another very sensitive parameter is the learning rate. If the learning rate is much too large, the reconstruction error during pre-training, usually increases dramatically and the weights may explode (become uncontrollable big). If the learning rate is reduced while the network is learning normally, the reconstruction error will usually fall significantly. This is not necessarily a good thing. It is due, in part, to the smaller noise level in the stochastic weight updates and it is generally accompanied by slower learning in the long term. Towards the end of learning, however, it typically pays to decrease the learning rate. Averaging the weights across several updates is an alternative way to remove some of the noise from the final weights.

Fine-tuning Phase

The fine-tuning phase is shown in the right part of Fig.9 in which an output labelling layer is added on top of the pre-trained DBN. In our work, these units correspond to the language-specific phonemes, although tied triphone states labels were also obtained for the more complex configurations. Each unit corresponds to the label of input features which converts a number of Bernoulli distributed units into a multinomial distribution through the softmax function (see ec.4). The BP algorithm is used to jointly tune all model parameters by minimizing the cross entropy function in eq.8

At this point of the experimentation it comes crucial to have well defined data sets to choose the metaparameters for the network. A wrong method is to try lots of alternatives and see which gives the best performance (lower classification error).

This is easy to do but it gives a false impression of how well the network works. The settings that work best on the test set are unlikely to work as well on a new test set drawn from the same distribution.

So here is a better way to choose metaparameters, divide the total data set into three subsets:

. Training data: is used for learning the parameters of the model.
. Validation data: is not used for learning but is used for deciding what settings of the metaparameters work best.
. Test data: is used to get a final unbiased estimate of how well the network works. We expect this estimate to be worst than on the validation data.

Learning is difficult in densely-connected, directed belief nets that have many hidden layers because it is difficult to infer the conditional distribution of the hidden activities when given a data vector.

5 Conclusions

When neural nets were first used, they were trained discriminatively. It was only recently that researchers showed that significant gains could be achieved by adding an initial stage of generative pre-training that somehow ignores the ultimate goal of the system.

The pre-training is much more helpful in deep neural nets than in shallow ones, especially when limited amounts of labeled training data are available. It reduces overfitting, and it also reduces the time required for discriminative fine-tuning with BP, which was one of the main impediments to using DNNs when neural networks were first used in place of GMMs in the 1990s. The successes achieved using pre-training led to a resurgence of interest in DNNs for acoustic modeling.

Retrospectively, it is now clear that most of the gain comes from using DNNs to exploit information in neighbouring frames and from modeling tied context-dependent states. Pre-training is helpful in reducing overfitting, and it does reduce the time taken for fine-tuning.

There is no reason to believe that we are currently using the optimal types of hidden units or the optimal network architectures, and it is highly likely that both the pre-training and fine-tuning algorithms

can be modified to reduce the amount of overfitting and the amount of computation. We therefore expect that the performance gap between acoustic models that use DNNs and ones that use GMMs will continue to increase for some time.

Currently, the biggest disadvantage of DNNs compared with GMMs is that it is much harder to make good use of large cluster machines to train them on massive data sets. This is offset by the fact that DNNs make more efficient use of data so they do not require as much data to achieve the same performance, but better ways of parallelizing the fine-tuning of DNNs is still a major issue.

Focusing in LID tasks, DBFs are generated from a specially structured deep neural network, in which one of the hidden layers - the bottleneck layer - has a small number of hidden units, compared to other layers. Although DBFs were shown to work well for the LID task, it is likely that the optimal DBF parameters for LID will differ to those used for ASR. If so, these still need to be explored.

Effective representation plays an important role in automatic LID. Recently, several representations that employ a pre-trained deep neural network as the front-end feature extractor, have achieved state-of-the-art performance. However the performance is still far from satisfactory for dialect and short-duration utterance identification tasks, due to the deficiency of existing representations. To address this issue, could be very promising to use the information extracted from different layers of the DNN structure and evaluate their influence or repercussion in the final objective: LID.

In statistical pattern recognition, if you want to recognize patterns, all that you have to do is to get some features from the sound signal or whatever the input is, and learn to weight the features to decide which class wins. But how do we decide what features to use?

Today deep neural networks achieve outstanding performance on many important problems in speech recognition, computer vision and natural language processing. However effective DNN training requires a lot of expertise. This report has attempted to summarize part of our practical knowledge acquired facing this challenge.

References

1. A., N.M.: Neural Networks and Deep Learning. Determination Press, Upper Saddle River, NJ, USA (2015)
2. Yu, D., Deng, L.: Automatic Speech Recognition - A Deep Learning Approach. Springer (October 2014)
3. Bengio, Y.: Practical recommendations for gradient-based training of deep architectures. Technical Report Arxiv report 1206.5533, Université de Montréal (2012)
4. Dahl, G.E., Yu, D., Deng, L., Acero, A.: Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. Trans. Audio, Speech and Lang. Proc. **20**(1) (jan 2012) 30–42
5. Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural Computation **18**(7) (2006) 1527–1554
6. Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Improving neural networks by preventing co-adaptation of feature detectors. CoRR **abs/1207.0580** (2012)
7. Bottou, L.: On-line learning in neural networks. Cambridge University Press, New York, NY, USA (1998) 9–42
8. Smolensky, P.: Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. MIT Press, Cambridge, MA, USA (1986) 194–281
9. Hinton, G.E.: Training products of experts by minimizing contrastive divergence. Neural Computation **14**(8) (2002) 1771–1800
10. Hinton, G.E.: A practical guide to training restricted boltzmann machines. In Montavon, G., Orr, G.B., Müller, K.R., eds.: Neural Networks: Tricks of the Trade (2nd ed.). Volume 7700 of Lecture Notes in Computer Science. Springer (2012) 599–619
11. Hinton, G., Deng, L., Yu, D., Dahl, G., rahman Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., Kingsbury, B.: Deep neural networks for acoustic modeling in speech recognition. Signal Processing Magazine (2012)
12. Mohamed, A., Dahl, G., Hinton, G.: Acoustic modeling using deep belief networks. Audio, Speech, and Language Processing, IEEE Transactions on **20**(1) (jan. 2012) 14 –22

13. Seide, F., Li, G., Yu, D.: Conversational speech transcription using context-dependent deep neural networks. In: in Proc. Interspeech '11. 437–440
14. Hermansky, H., Ellis, D.W., Sharma, S.: Tandem connectionist feature extraction for conventional hmm systems. In: Acoustics, Speech, and Signal Processing, 2000. ICASSP'00. Proceedings. 2000 IEEE International Conference on. Volume 3., IEEE (2000) 1635–1638
15. Grézl, F., Karafiát, M., Kontar, S., Cernocký, J.: Probabilistic and bottle-neck features for LVCSR of meetings. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2007, Honolulu, Hawaii, USA, April 15-20, 2007. (2007) 757–760
16. Trentin, E., Gori, M.: A survey of hybrid ann/hmm models for automatic speech recognition. Neurocomputing (2001) 91–126
17. Morgan, N., Bourlard, H.: Continuous speech recognition using multilayer perceptrons with hidden Markov models (1990)
18. Grezl, F., Fousek, P.: Optimizing bottle-neck features for lvcsr. In: Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on. (March 2008) 4729–4732
19. Sugiyama, M.: Automatic language recognition using acoustic features. In: Proceedings of the Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference. ICASSP '91, Washington, DC, USA, IEEE Computer Society (1991) 813–816
20. Hazen, T.: Automatic language identification using a segment-based approach. Technical report, Cambridge, MA, USA (1993)
21. Zissman, M.A.: Comparison of four approaches to automatic language identification of telephone speech. IEEE Transactions on Speech and Audio Processing **4**(1) (1996) 31
22. Matrouf, D., Adda-Decker, M., Lamel, L., Gauvain, J.L.: Language identification incorporating lexical information. In: ICSLP, ISCA (1998)
23. Kirchhoff Katrin, Parandekar Sonia, B.J.: Mixed-memory markov models for automatic language identification. In: ICASSP. (2002) 761–764
24. Siniscalchi, S.M., Reed, J., Svendsen, T., Lee, C.H.: Exploring universal attribute characterization of spoken languages for spoken language recognition. In: INTERSPEECH. (2009) 168–171
25. Martin, A.F., Greenberg, C.S.: The 2009 NIST language recognition evaluation. In: Odyssey 2010: The Speaker and Language Recognition Workshop, Brno, Czech Republic, June 28 - July 1, 2010. (2010) 30
26. Torres-Carrasquillo, P.A., Singer, E., Gleason, T.P., McCree, A., Reynolds, D.A., Richardson, F., Sturim, D.E.: The MITLL NIST LRE 2009 language recognition system. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2010, 14-19 March 2010, Sheraton Dallas Hotel, Dallas, Texas, USA. (2010) 4994–4997
27. Torres-Carrasquillo, P.A., Singer, E., Kohler, M.A., Greene, R.J., Reynolds, D.A., Jr., J.R.D.: Approaches to language identification using gaussian mixture models and shifted delta cepstral features. In: Proc. International Speech Communication Association Conference. (2002)
28. Dehak, N., Torres-Carrasquillo, P.A., Reynolds, D.A., Dehak, R.: Language recognition via i-vectors and dimensionality reduction. In: Proc. International Speech Communication Association Conference. (2011) 857–860
29. Yu, D., Seltzer, M.L.: Improved bottleneck features using pretrained deep neural networks. In: INTERSPEECH 2011, 12th Annual Conference of the International Speech Communication Association, Florence, Italy, August 27-31, 2011. (2011) 237–240
30. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science **313**(5786) (Jul 2006) 504–507
31. Jiang, B., Song, Y., Wei, S., Liu, J.H., McLoughlin, I.V., Dai, L.R.: Deep bottleneck features for spoken language identification. PLoS ONE (2014)
32. Song, Y., Jiang, B., Bao, Y., Wei, S., Dai, L.R.: I-vector representation based on bottleneck features for language identification. Electronics Letters **49**(24) (2013) 1569 – 1570
33. Yu, D., Deng, L., Dahl, G.E.: Roles of pre-training and fine-tuning in context-dependent dbn-hmms for real-world speech recognition. In: NIPS 2010 workshop on Deep Learning and Unsupervised Feature Learning. (December 2010)
34. LeCun, Y., Bottou, L., Orr, G.B., Müller, K.R.: Effiicient backprop. In: Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop, London, UK, UK, Springer-Verlag (1998) 9–50

Annexes

1 Activation Function

Some of the more used activation functions:

Table 1. Most used activation functions.

Name	Function
Lineal	$f(z) = z$
Sigmoid or logistic	$\sigma(z) = \frac{1}{1+e^{-z}}$
Tanh	$tanhz = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
Step	$fz = \begin{cases} -1 \text{ if } z < 0, \\ 1 \text{ if } z \geq 0. \end{cases}$
RLU	$f(z) = max(0,z)$

The sigmoidal activation function is the most commonly used in NN, it enables much more versatile learning algorithms. A sigmoidal neuron feeds the weighted sum of the inputs into the logistic function which returns a value between 0 and 1. When the weighted sum is very negative, the returned value is very close to 0, when the weighted sum is very large and positive, the return value is very close to 1. The logistic function is a good choice because it has a nice looking derivative, which makes learning a simpler process.

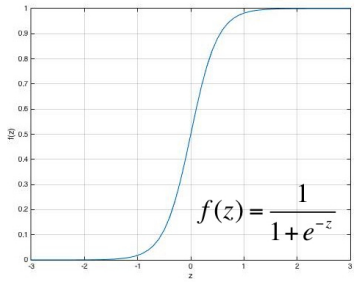

Fig. 10. Sigmoid function.

Since the $tanh(z)$ function is a rescaled version of the sigmoid function, these two activation functions have the same modelling power. However, the output range of $\sigma(z)$ is (0, 1) which encourages sparse but, at the same time, asymmetric activation values. On the other hand, the output range of $tanh(z)$ is $(-1,+1)$, and thus the activation value is symmetric, which was believed to help the model training [34]. Another popular activation function is the rectified linear unit (ReLU) function which enforces sparse activations [4] and has very simple gradient. Since sigmoid function is still the most popular activation functions used, in all the following discussions we assume that the sigmoid activation function is used unless otherwise noted.

The softmax function, or normalized exponential, is a generalization of the logistic function that squashes the C-dimensional vector $\mathbf{z}^L$ of the outputs of the layer L to a C-dimensional vector $\sigma(\mathbf{z})$ of

[4] The output of the sigmoid function can be very close to 0 but cannot reach 0, while the output of the ReLU function can be exactly 0.

real values in the range (0, 1) that add up to 1. The function is given by

$$\text{softmax}_i\left(\mathbf{z}^L\right) = \frac{e^{z_i^L}}{\sum_{j=1}^{C} e^{z_j^L}}. \tag{21}$$

In neural network simulations, the softmax function is often implemented at the final layer of a network used for classification. Such networks are then trained under a log loss (or cross-entropy) regime, giving a non-linear variant of multinomial logistic regression.

2 Toolbox for DNN Implementation

State-of-the-art DNNs can have from millions to well over one billion parameters to adjust via back-propagation. Furthermore, DNNs require a large amount of training data to achieve high accuracy, meaning hundreds of thousands to millions of input samples will have to be run through both a forward and backward pass. Because neural networks are created from large numbers of identical neurons they are highly parallel by nature. This parallelism maps naturally to GPUs, which provide a significant speed-up over CPU-only training.

The success of DNNs has been greatly accelerated by using GPUs, which have become the platform of choice for training large, complex DNN-based systems.

Which are the best available toolboxes for DNN implementation, mainly for speech applications?

- Theano: CPU/GPU symbolic expression compiler in python (from LISA lab at University of Montreal) `http://deeplearning.net/software/theano/` Pros: support for symbolical computation, tutorials and great community, build-in support for GPU.
  - keras: Theano-based Deep Learning library `http://keras.io/`
  - PDNN: is a Python deep learning toolkit developed under the Theano environment `http://www.cs.cmu.edu/~ymiao/pdnntk.html`.
- Kaldi: is a complete speech recognition software, which also embeds a deep neural network tool kit (not as flexible as Theano) `http://kaldi.sourceforge.net/index.html`. It uses CUDA matrix library to access to GPU-based operations.
- Matlab Deep Learning Toolbox: Deep Belief Nets, Stacked Autoencoders, Convolutional Neural Nets. See also Ruslan Salakhutdinov implementations `http://www.cs.toronto.edu/~rsalakhu/code.html` Only CPU version, slow.
  Most academic researchers in the field of deep learning rely on Theano, the grand-daddy of deep-learning frameworks, which is written in Python.
  Numerous open-source deep-libraries have been built on top of Theano, including Keras, Lasagne and Blocks. These libs attempt to layer an easier to use API on top of Theano's occasionally non-intuitive interface.

A Other Frameworks (designed for computer vision)

- **Pylearn2 Vision**: is a machine learning library, is still undergoing rapid development. Most of its functionality is built on top of Theano `http://deeplearning.net/software/pylearn2/`. Pylearn2 is generally considered the library of choice for neural networks and deep learning in python. Its designed for easy scientific experimentation rather than ease of use, so the learning curve is rather

steep, but if you take your time and follow the tutorials I think you'll be happy with the functionality it provides. Everything from standard Multilayer Perceptrons to Restricted Boltzmann Machines to Convolutional Nets to Autoencoders are provided. There's great GPU support and everything is built on top of Theano, so performance is typically quite good. The source for Pylearn2 is available on github. Be aware that Pylearn2 has the opposite problem of pybrain at the moment – rather than being abandoned, Pylearn2 is under active development and is subject to frequent changes.
Pylearn2 is a machine-learning library, while Theano is a library that handles multidimensional arrays, like Numpy. Both are powerful tools widely used for research purposes and serving the large Python community. They are well suited to data exploration and explicitly state that they are intended for research.

- **Caffe**: is a well-known and widely used machine-vision library that ported Matlab's implementation of fast convolutional nets to C and C++. Caffe is not intended for other deep-learning applications such as text, sound or time series data. Both Deeplearning4j and Caffe perform image classification with convolutional nets, which represent the state of the art. In contrast to Caffe, Deeplearning4j offers parallel GPU support for an arbitrary number of chips, as well as many, seemingly trivial, features that make deep learning run more smoothly on multiple GPU clusters in parallel. While it is widely cited in papers, Caffe is chiefly used as a source of pre-trained models hosted on its Model Zoo site. Deeplearning4j is building a parser to import Caffe models to Spark.
- **Torch**: an open source development environment for numerics, machine learning, and computer vision, with a particular emphasis on deep learning and convolutional nets. `http://torch.ch/`. It uses an underlying C/CUDA implementation. Torch is widely used at a number of academic labs as well as at Google/DeepMind, Twitter, NVIDIA, AMD, Intel, and many other companies.
- **cuDNN**: (NVIDIA CUDA DNN library): accelerates widely-used deep learning frameworks (available to CUDA Registered Developers) `https://developer.nvidia.com/cudnn`.

B Deep Learning Projects based on Theano

Much of the deep learning community is focused on Python. After all, Python has great syntactic elements that allow you to add matrices together without creating explicit classes. Likewise, Python has an extensive scientific computing environment with native extensions like Theano and Numpy.

Keras, Blocks and Lasagne all seem to share the same goal of being more libraries than framework. It can be used only one part (e.g. a Layer implementation, training algorithm) without having to pull in everything :

- Lasagne: `https://github.com/Lasagne/Lasagne`, `http://lasagne.readthedocs.org/en/latest/`
- Keras: `https://github.com/fchollet/keras`
- Blocks: `https://github.com/mila-udem/blocks`
- Deepy: `https://github.com/uaca/deepy`
- Nolearn: `https://github.com/dnouri/nolearn`

B.1 Keras vs. Lasagne

Lasagne is being built by a team of deep learning and music information retrieval researchers. Keras seems to share a lot of design goals with this project, but there are also some significant differences [5].

[5] Taken from `https://news.ycombinator.com/item?id=9283105`

Both want to build something that's minimalistic, with a simple API, and that allows for fast prototyping of new models. Keras seems to be built "on top of" Theano in the sense that it hides all the Theano code behind an API (which looks almost exactly like the Torch7 API).

Lasagne is built to work "with" Theano instead. It does not try to hide the symbolic computation graph, because the creators believe that is where Theano's power comes from. The library provides a bunch of primitives (such as Layer classes) that make building and training neural networks a lot easier. Lasagne creators are also specifically aiming at extensibility: the code is readable and it's really easy to implement the layer classes.

Another difference seems to be the way the concept of a "layer" is interpreted: a layer in Lasagne adheres as closely as possible to its definition in literature. Keras (and Torch7) treat each operation as a separate stage instead, so a typical fully connected layer has to be constructed as a cascade of a dot product and an element wise non-linearity.

Layers are also first-class citizens in Lasagne, and a model is usually referred to simply by its output layer or layers. There is no separate "Model" class because they want to keep the interface as small as possible and so far they've done fine without it. In Keras (and Torch7) the layers cannot function by themselves and need to be added to a model instance first.

For now, all Lasagne really does in the end is make it easier to construct Theano expressions - doesn't have any tools for iterating through datasets for example. They plan to rely heavily on Python generators for this. The scikit-learn like "model.fit(X, y)" paradigm, which Keras also seems to use, only really works for small datasets which fit in memory. For larger datasets, they recommend generators as the way to go. Incidentally, Nolearn (`https://github.com/dnouri/nolearn`) provides a wrapper for Lasagne models with a scikit-learn like interface. Lasagne is not released yet - the interface is not 100% stable yet, and documentation and tests are a work in progress (although both are progressing nicely). But a lot of people have started using it already, they've built up a nice user base and a lot of people have started contributing code as well.

A non-exhaustive list of their design goals for the library is in the README on their GitHub page: https://github.com/benanne/Lasagne

RT_082, abril 2016
Aprobado por el Consejo Científico CENATAV

Editor: Lic. Lucía González Bayona
Diseño de Portada: Di. Alejandro Pérez Abraham
RNPS No. 2142
ISSN 2072-6287
Indicaciones para los Autores:
Seguir la plantilla que aparece en www.cenatav.co.cu
C E N A T A V
7ma. A No. 21406 e/214 y 216, Rpto. Siboney, Playa;
La Habana. Cuba. C.P. 12200
Impreso en Cuba