Contents

Contents **i**

List of Figures **iii**

1 Deep Neural Networks **1**

1.1 History and Motivation 1

1.2 Constructing DNNs: the Multilayer Perceptron 1

1.2.1 Architecture of the Deep Multilayer Perceptron 1

1.2.2 Training and related issues 4

1.3 Pretraining 12

1.4 Recurrent Neural Networks 21

1.4.1 State-Space formulation of the basic RNN 22

1.4.2 Training 23

List of Figures

1.1 *A deep neural network with three hidden layers [YD14].* 2
1.2 *Computation of DNN output [YD14].* 4
1.3 *Back-propagation algorithm [YD14].* 5
1.4 *An example of a Restricted Boltzmann Machine [YD14].* 14
1.5 *Marginal probability density function of a Gaussian-Bernoulli RBM [YD14].* 16
1.6 *Contrastive Divergence algorithm for RBM training [YD14].* 17
1.7 *Different perspectives of the same RBM [YD14].* 18
1.8 *Discriminative pretraining [YD14].* 20
1.9 *Backpropagation-through-time [YD14].* 24

Chapter 1

Deep Neural Networks

In this chapter we will give a general introduction to deep neural networks (*DNNs*). We will begin by presenting the ideas and motivation behind deep architectures and then move on to describe a few of these models as well as practical issues concerning building and training such models.

1.1 History and Motivation

1.2 Constructing DNNs: the Multilayer Perceptron

[YD14]

Although initially the term "deep neural network" was used to identify a multilayer perceptron with many hidden layers, it has now come to name a whole class of neural network models having a "deep" architecture, i.e. having at least three hidden layers. However, we will use the conventional multilayer perceptron to present the basic deep architecture and cover the training issues associated with it.

1.2.1 Architecture of the Deep Multilayer Perceptron

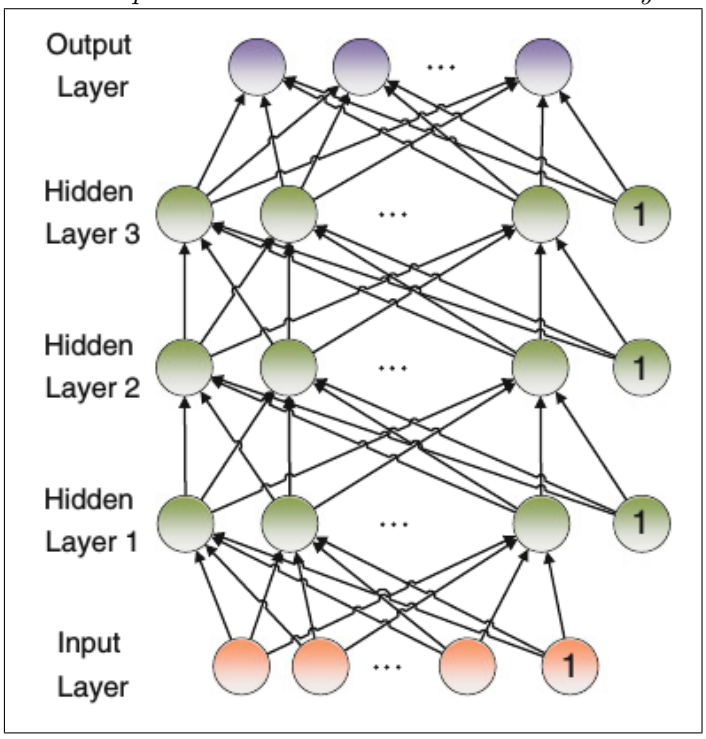
As already mentioned, the simplest deep neural network is a multilayer perceptron with at least three hidden layers, that is, excluding the input and output layers. A DNN with a total of five layers can be seen in figure 1.1

We can denote the input layer as layer 0 and the output layer as layer L for an $L+1$-layer DNN.

The neurons in the first layer correspond to each feature (or dimension) in the input vector, i.e.

$$\boldsymbol{v}^0 = \boldsymbol{x}_{input}$$

Figure 1.1: *A deep neural network with three hidden layers [YD14].*



whereas for the first L layers the output is calculated as follows:

$$\boldsymbol{v}^l = f(\boldsymbol{z}^l) = f\Big(\boldsymbol{W}^l \boldsymbol{v}^{l-1} + \boldsymbol{b}^l\Big),\ 0 < l < L$$

where

$$\boldsymbol{z}^l = \boldsymbol{W}^l \boldsymbol{v}^{l-1} + \boldsymbol{b}^l \in \mathcal{R}^{N_l \times 1}$$

$$\boldsymbol{v}^l \in \mathcal{R}^{N_l \times 1}$$

$$\boldsymbol{W}^l \in \mathcal{R}^{N_l \times N_{l-1}}$$

$$\boldsymbol{b}^l \in \mathcal{R}^{N_l \times 1}$$

and $N_l \in \mathcal{R}$, are respectively the excitation vector, the activation vector, the weight matrix, the bias vector and the number of neurons at layer l.

The function $f(\cdot) : \mathcal{R}^{N_l \times 1} \to \mathcal{R}^{N_l \times 1}$ is the activation function of the corresponding neuron that is applied element-wise to the excitation vector. The most common activation function is the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

followed by the second most common, the hyperbolic tangent

$$\tanh(z) = \frac{e^z + e^{-z}}{e^z + e^{-z}}$$

which is a rescaled version of the former. Both have the same modeling power, but the output range of the hyperbolic tangent is (-1,+1) , therefore the activation values are symmetric and this was believed to help during training. On the other hand the sigmoid function has an output range of (0,1), which stimulates sparse but asymmetric activation values. Recently, another type of activation function, the rectified linear unit (ReLU) has been suggested:

$$ReLu(z) = max(0, z)$$

ReLu has been gaining in popularity since it enforces sparse activation values and has a simple gradient:

$$\nabla_{ReLu}(z) = max(0, sgn(z))$$

As far as the output layer is concerned, its form depends on the task of interest.

In case of regression tasks, a linear layer is used to generate the output vector $\boldsymbol{v}^L \in \mathcal{R}^{N_l}$, where N_l is the output dimension:

$$\boldsymbol{v}^L = \boldsymbol{z}^L = \boldsymbol{W}^L \boldsymbol{v}^{L-1} + \boldsymbol{b}^L$$

In case of classification tasks, each output neuron represents a class $i \in 1, ..., C$. where $C = N_L$ is the number of classes. In this case, the output of the DNN will be a probability distribution over the possible classes. The value of the i^{th} output neuron represents the probability $P_{dnn}(i|\boldsymbol{x}_i)$ that the input vector belongs to class i.

Since we want the output vector to be a valid probability distribution we impose the following constraints on it

$$v_i^L \geq 0$$

and

$$\sum_{i=1}^{C} v_i^L = 1$$

and use as output function the softmax function:

$$v_i^L = P_{dnn}(i|\boldsymbol{x}_i) = softmax(\boldsymbol{z}^L) = \frac{e^{z_i^L}}{\sum_{j=1}^{C} e^{z_j^L}}$$

where z_i^L is the i^{th} element of the excitation vector $\boldsymbol{z}^L$.

Given an input matrix with observations $\boldsymbol{O}$ the output of the DNN can be computed as described in figure 1.2

Figure 1.2: *Computation of DNN output [YD14].*

```
1: procedure FORWARDCOMPUTATION(O)
                                        ▷ Each column of O is an observation vector
2:     V^0 ← O
3:     for ℓ ← 1; ℓ < L; ℓ ← ℓ + 1 do        ▷ L is the total number of layers
4:         Z^ℓ ← W^ℓ V^{ℓ−1} + B^ℓ          ▷ Each column of B^ℓ is b^ℓ
5:         V^ℓ ← f(Z^ℓ)        ▷ f(.) can be sigmoid, tanh, ReLU, or other functions
6:     end for
7:     Z^L ← W^L V^{L−1} + B^L
8:     if regression then                                  ▷ regression task
9:         V^L ← Z^L
10:    else                                              ▷ classification task
11:        V^L ← softmax(Z^L)                ▷ Apply softmax column-wise
12:    end if
13:    Return V^L
14: end procedure
```

1.2.2 Training and related issues

Given that a multilayer perceptron with a sufficiently large hidden layer can approximate any function ([HSW89]), it is evident that a multilayer perceptron with more than one hidden layers can serve as a universal approximator as well, that is, compute any mapping

$$g : \mathcal{R}^D \rightarrow \mathcal{R}^C$$

from the input space $\mathcal{R}^D$ to the output space $\mathcal{R}^C$. Given a set of training samples we can estimate the network parameters $\{\boldsymbol{W}, \boldsymbol{b}\}$ with a training process, characterized by a training criterion and a learning algorithm.

Training criterion

The training criterion is chosen so that an improvement in it reflects an improvement in the final evaluation score. Therefore, it should somehow express the final goal of the task, yet at the same time be easy to evaluate.

Ideally, we would like to estimate the DNN parameters that minimize the expected cost:

$$J_E = \mathrm{E}(J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}_i, \boldsymbol{y}))$$

where J is the cost (or loss) function given the model parameters $\{\boldsymbol{W}, \boldsymbol{b}\}$ together with the input vector $\boldsymbol{x}_i$ and the corresponding output $\boldsymbol{y}$. However, since we cannot know in advance the output for unseen samples, we can only optimize empirical criteria based on the training set.

The two most popular empirical criteria used in DNN training are the mean square error (MSE) for regression tasks and the cross-entropy (CE) for classification

Figure 1.3: *Back-propagation algorithm [YD14].*

procedure BACKPROPAGATION($\mathbb{S} = \{(\mathbf{o}^m, \mathbf{y}^m) \,|0 \leq m < M\}$)
▷ $\mathbb{S}$ is the training set with M samples
  Randomly initialize $\left\{\mathbf{W}_0^\ell, \mathbf{b}_0^\ell\right\}, 0 < \ell \leq L$ ▷ L is the total number of layers
  while Stopping Criterion Not Met **do**
    ▷ Stop if reached max iterations or the training criterion improvement is small
    Randomly select a minibatch $\mathbf{O}$, $\mathbf{Y}$ with M_b samples.
    Call ForwardComputation($\mathbf{O}$)
    $\mathbf{E}_t^L \leftarrow \mathbf{V}_t^L - \mathbf{Y}$ ▷ Each column of $\mathbf{E}_t^L$ is $\mathbf{e}_t^L$
    $\mathbf{G}_t^L \leftarrow \mathbf{E}_t^L$
    for $\ell \leftarrow L; \ell > 0; \ell \leftarrow \ell - 1$ **do**
      $\nabla_{\mathbf{W}_t^\ell} \leftarrow \mathbf{G}_t^\ell \left(\mathbf{v}_t^{\ell-1}\right)^{\mathrm{T}}$
      $\nabla_{\mathbf{b}_t^\ell} \leftarrow \mathbf{G}_t^\ell$
      $\mathbf{W}_{t+1}^\ell \leftarrow \mathbf{W}_t^\ell - \frac{\varepsilon}{M_b} \nabla \mathbf{W}_t^\ell$ ▷ Update $\mathbf{W}$
      $\mathbf{b}_{t+1}^\ell \leftarrow \mathbf{b}_t^\ell - \frac{\varepsilon}{M_b} \nabla \mathbf{b}_t^\ell$ ▷ Update $\mathbf{b}$
      $\mathbf{E}_t^{\ell-1} \leftarrow \left(\mathbf{W}_t^\ell\right)^{\mathrm{T}} \mathbf{G}_t^\ell$ ▷ Error backpropagation
      if $\ell > 1$ **then**
        $\mathbf{G}_t^{\ell-1} \leftarrow f'\left(\mathbf{Z}_t^{\ell-1}\right) \bullet \mathbf{E}_t^{\ell-1}$
      end if
    end for
  end while
  Return $dnn = \left\{\mathbf{W}^\ell, \mathbf{b}^\ell\right\}, 0 < \ell \leq L$
end procedure

tasks, which are defined as follows:

$$J_{MSE}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}_{i,train}, \boldsymbol{y}_{o,train}) = \frac{1}{2M} \sum_{m=1}^{M} \|\boldsymbol{v}^L - \boldsymbol{y}\|^2$$

and

$$J_{CE}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}_{i,train}, \boldsymbol{y}_{o,train}) = \frac{1}{M} \sum_{m=1}^{M} (-\sum_{i=1}^{C} y_i \log(v_i^L))$$

where y_i and v_i in J_{CE} are respectively, the probability observed in the training set that $\boldsymbol{x}$ belongs to class i ($y_i = P_{empirical}(i|\boldsymbol{x})$) and the same probability estimated from the DNN $v_i^L = P_{DNN}(i|\boldsymbol{x})$).

The minimization of the cross-entropy criterion is equivalent to minimizing the the Kullback-Leibler divergence between the empirical probability distribution and the probability distribution estimated from the deep neural network.

Training algorithm and practical considerations

Having selected an appropriate training criterion, we can train the network using the well known back-propagation algorithm which is based on the chain rule for gradient computation. The algorithm is summarized in figure 1.3.

Despite the simplicity of the algorithm, its application to DNNs gives rise to many problems, the most important ones being the speed of convergence and the vanishing or exploding gradients that get in the way of the training. It is therefore

crucial to address these issues by specifically tuning the DNN and the training set to avoid such drawbacks.

Data preprocessing Data preprocessing is an important part of the pre-training phase in the DNN construction. There are two principal ways of preprocessing each one attacking a different problem that might arise during training.

The first preprocessing technique is referred to as per-sample feature normalization and it results in reducing the variability in the final feature presented to the DNN. This is desired since we want to drop any variation that is not important to the final decision made by the model and would otherwise add extra difficulties both at the output stage as well as during training, as the network would try to learn too much redundant information. For example, in image recognition, it is desired to reduce the variability introduced by the brightness, or in speech recognition we want to avoid acoustic channel distortions. In this last case we can achieve our goal by calculating the per-utterance mean for each feature and subtracting it from all frames in the utterance (cepstral mean normalization).

The second preprocessing technique is the global feature standardization and it aims to scale the data along each dimension with a global transformation so that the final data vectors lie in a similar range of numerical values. The global transformation is estimated from the training data and is then applied to both training and test sets. Global feature standardization will improve performance in later steps of the training process. As far as DNN training is concerned, it allows us to use the same learning rate across all weight dimensions and still achieve good performance.

A common global transformation in speech recognition is to standardize each dimension of the feature vector to have zero mean and unit variance. If we use MFCC vectors without standardization, the energy component will dominate the learning procedure since its values are much greater than the rest of the coefficients.

Initialization of model parameters The back-propagation algorithm starts given a set of initial network parameters. Since the DNN is highly non-linear and the training criterion with respect to the model parameters is non-convex, initialization of the model parameters can greatly affect the end model. There are many suggestions for the initialization of a DNN model, all of which are based on two assumptions.

First, the weights should be initialized so that each neuron operates in the linear range of the activation function (e.g. sigmoid) at the start of the learning. If the weights were very large, neurons would saturate, i.e. take an extreme value (close to zero or one in the case of a sigmoid function) leading to vanishing gradients. This

becomes evident in the case of the sigmoid as we can observe from its derivative:

$$\sigma'(\boldsymbol{z}_t^l) = (1 - \sigma(\boldsymbol{z}_t^l))\sigma(\boldsymbol{z}_t^l) = (1 - \boldsymbol{v}_t^l)\boldsymbol{v}_t^l$$

On the other hand, if the neurons operate in the linear range, the gradients are large enough and learning can proceed smoothly. It is important to point out that the excitation vector depends on both the weights and the input values. If we have standardized the input values in the preprocessing stage, then the weight initialization can become easier.

Second, the weights should be initialized at random. Since neurons in DNNs are symmetric and interchangeable, random initialization serves the purpose of symmetry breaking. If all neurons were initialized with the same values then they would have the same output thus detect the same patterns in the lower layers.

Bengio et. al. ([GB10]) have suggested that the weight values should be uniformly sampled from an interval depending on the activation function. For the sigmoid function they suggested the interval

$$\Big[-4\sqrt{\frac{6}{fan_{in} + fan_{out}}}, 4\sqrt{\frac{6}{fan_{in} + fan_{out}}} \Big]$$

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. This approach is based on the idea that neurons with more inputs should have smaller weights ([Ben12]).

LeCun et. al. ([LBOM98]) suggested to draw the initial weight values for layer l from a zero-mean Gaussian distribution with standard deviation $\sigma_{\boldsymbol{W}^{l+1}} = \frac{1}{\sqrt{N_l}}$, where N_l is the number of connections feeding into the node.

When it comes to speech recognition applications, usually each DNN layer has 1000-2000 neurons and initializing the weights either from a Gaussian distribution $\mathcal{N}(w; 0, 0.05)$ or from a uniform distribution in the range of [-0.05,0.05] we can achieve good results.

As far as the bias vectors $\boldsymbol{b}^l$ are concerned, they can be initialized to zero.

Regularization Taking into consideration the huge number of parameters in a DNN it is easy to understand the risk of overfitting the model to the training data. Overfitting occurs because although we aim to minimize the expected loss, we actually minimize the empirical loss defined on the training set. To control overfitting we try to affect the model parameters through the training function so that they do not fit the training data too well. This process is known as regularization or weight decay. Regularization aims to help the model learn patterns often seen in the training data yet not learn every noise peculiarity present in the training set ([Nie15]).

The most common forms of regularization are based on the L_1 and L_2 norms respectively:

$$R_1(\boldsymbol{W}) = \|vec(\boldsymbol{W})\|_1 = \sum_{l=1}^{L} \|vec(\boldsymbol{W}^l)\|_1$$

$$R_2(\boldsymbol{W}) = \|vec(\boldsymbol{W})\|_2^2 = \sum_{l=1}^{L} \|vec(\boldsymbol{W}^l)\|_2^2$$

Including the regularization terms the training criterion becomes

$$J_R(\boldsymbol{W}, \boldsymbol{b}; \mathcal{S}_{train}) = J(\boldsymbol{W}, \boldsymbol{b}; \mathcal{S}_{train}) + \lambda R(\boldsymbol{W})$$

where λ is an interpolation weight called regularization weight. Regularization is particularly helpful when the training set size is small compared to the number of parameters of the DNN. For DNNs used in speech recognition, there are usually millions of parameters in the model and the regularization weight should be small ($\approx 10^{-4}$) or zero when the training set is large.

In Bayesian context, the regularization term is the negative log-prior $-logP(\theta)$ on the parameters θ. Adding the regularization term makes the training criterion correspond to the negative joint likelihood of training data and parameters

$$-logP(data, \theta) = -logP(data|\theta) - logP(\theta)$$

with the loss function $L(z, \theta)$ being interpreted as $-logP(z|\theta)$ and $-logP(data|\theta) = -\sum_{t=1}^{T} L(z_t, \theta)$, where $z_t,\ t = 1, ..., T$ are the training data. In case we are training our model with a stochastic gradient based method, as is usually the case, we want to use an unbiased estimation of the gradient of the total training criterion including both the loss function and the regularization term. In a mini-batch or online learning approach it is not trivial to include the regularization in the above sum, since it is different from the sum of the loss function on all examples and the regularization term. In these cases, the regularization term should be properly weighted not only with λ but also with the inverse of the number of updates needed to see all the training data ([Ben12]).

Another popular regularization method is dropout. The basic idea behind dropout is to randomly omit a certain percentage (e.g. p) of the neurons in each hidden layer for each presentation of the samples during training. This would mean that each random combination of the remaining neurons needs to perform well during training, which implies that each neuron depends less on other neurons to detect patterns.

One way to look at dropout is as a technique to add random noise to the training data, since each neuron takes input from a random collection of the neurons in the previous layer. Consequently, the excitation value will be different even if the same

input is fed to the DNN. The improvement in generalization comes from the fact that the capacity of the DNN is reduced since some weights must be dedicated to removing the noise inserted by dropout.

A different perspective of dropout is as a *bagging* technique. *Bagging* refers to the combination of different classifiers, each one performing better than the rest at identifying a particular pattern in the data. The resemblance of dropout to bagging comes from the fact that omitting neurons creates in essence a different layer, thus a different classifier, during each feed-forward pass. After multiple passes it is as if we are averaging the outputs of these "different" classifiers to produce the final output of our model.

It should be noted that regularization affects only the weights and not the biases of the layers. This is because having a large bias does not make the network as sensitive to its input as having large weights would.

Batch size The term *batch size* refers to the number of samples we observe before we make an update to the model parameters. This number will affect the convergence speed as well as the end model.

The simplest choice of batch size is the whole training set. This approach, also known as batch training, has several advantages: it has good convergence properties, there are many techniques that can speed it up like conjugate gradient ([HS52]) and L-BFGS ([LN89]) and it allows for easy parallelization. On the other hand, seeing the whole dataset before a parameter update, is prohibitive for large datasets, which are often met in speech processing tasks.

Another approach to setting the batch size is to set it equal to one. In this approach, commonly referred to as stochastic gradient descent (*SGD*) or online learning, we update the model parameters after seeing just a single sample from the training set. If the sample is independently and identically distributed, it can be shown that the gradient of the training criterion estimated from it is an unbiased estimation of the gradient on the whole training set, despite the fact that it is noisy. This noise is the advantage of the SGD over the batch training approach. Since deep neural networks are highly non-linear and non-convex, the loss function contains many local minima, many of which are poor. Batch learning will converge to the minimum of whatever basin the initial model parameters are in and therefore it makes the model highly dependent on its initialization. On the other hand, the noise present in the gradient estimation of the SGD approach can help the algorithm move away from a poor local minimum and go closer to a better one. This property is reminiscent of simulated annealing, which allows the model parameters to move in a direction locally poorer but globally better ([KGV83]).

Moreover, SGD often converges much faster than batch learning, especially in

large datasets. This is because first, it does not waste time in redundant samples in the dataset as batch learning does, and second because in each update the new gradient is estimated based on the new model rather than the old. This means that SGD moves faster towards the best model.

However, SGD cannot be easily parallelized and it will fluctuate around the local minimum due to the noise present in the gradient estimation, without fully converging to it. This fluctuation, can be sometimes useful, as it reduces overfitting.

The third approach to batch size is to set it to a number, usually small, between one and the training set size. This approach, called minibatch training, estimates the gradient based on a small number (a batch) of *randomly* drawn training samples. It must be noted that drawing the samples at random is crucial to the minibatch and the SGD approaches to training.

Similar to SGD, the minibatch-estimated gradient is also unbiased, yet the variance of the estimation is smaller than that in the SGD algorithm. Furthermore, we can easily parallelize the computations within the minibatch, which makes this approach converge faster than SGD.

Care should be taken as to how to choose the appropriate minibatch size, yet fortunately it can be chosen independently of the rest of the learning parameters. It can be determined automatically considering the variance of the gradient estimation, or it can be empirically determined by searching on a small subset of samples in each epoch. As far as speech recognition tasks are concerned, a minibatch size of 64-256 is good for the early stages of the training, which can be expanded to 1024-8096 in later stages.

Momentum In neural network training adding a momentum term to the update of the model parameters is known to improve the convergence speed. This is due to the fact that the parameters' updates will be based on all the previous gradients instead of only the current one and the oscillation problems commonly seen in the back-propagation algorithm will be reduced.

Learning rate and stopping criterion One of the most difficult to choose parameters in DNN training, yet maybe the most crucial, is the learning rate, as it can determine whether the training will convergence to an optimum, as well as the convergence speed. Most learning rate strategies choose an appropriate initial learning rate and modify it during training. The modifications are based on a strategy usually defined empirically.

A common practice to determine the initial optimal learning rate is to set it to the largest value that does not make the training criterion diverge. Consequently, one can start with a large value, observe the development of the training criterion and if it diverges, try setting the learning rate to a value three times smaller than

the initial and proceed likewise until there is no divergence ([Ben12]).

In the case of batch learning, an empirical suggestion for a learning strategy is to decrease the learning rate when the weight vectors oscillate and increase it when the weights follow a relatively steady course. However, this approach cannot be applied to SGD learning since the weights continuously fluctuate ([LBOM98]).

It has been proven that when the learning rate is set to

$$\epsilon = \frac{c}{t}$$

where t is the number of samples seen by the network and c is a constant, or generally

$$\epsilon_t = \frac{\epsilon_0 \tau}{max(t, \tau)}$$

which maintains a constant learning rate for the first τ steps and then decreases it in $\mathcal{O}(\frac{1}{t})$, SGD converges asymptotically, although the convergence will be slow since ϵ will quickly become too small ([YD14],[LBOM98]).

Network Architecture Network architecture, meaning the number of hidden layers and the number of neurons in each layer, is an important model parameter that can affect the computational complexity and consequently the training speed, as well as the performance of the end model itself.

Each layer in the network works in essence as a feature extractor of the previous layer. Consequently, one must make sure that the number of neurons in it is large enough to capture the patterns of interest of the input vector. This becomes especially important in the case of the lower layers, since lower layer features are more variable, and more neurons are required to model the hidden patterns. However, having too many neurons encloses the danger of overfitting. As a rule of thumb, the wide, shallow networks are easier to overfit whereas the narrow, deep networks are easier to underfit to the training data.

Taking these into consideration, it is often the case that a different number of neurons is used in each layer, with wide layers with many neurons being placed close to the network input and narrow layers with less neurons being placed close to the output. If however all layers have the same number of neurons, simply adding more layers in some cases might cause the network to underfit since the extra layers add extra constraints to the model parameters.

On the other hand, Bengio ([Ben12]) reports that networks using same-size layers perform better or the same as using decreasing or increasing size layers. He mentions though, that this could depend on the data used.

Regarding speech recognition tasks, it has been found that 5-7 layer DNNs with 1000-3000 neurons per layer work well.

1.3 Pretraining

Considering the above discussion and the difficulties in DNN training, it must have already been clear that the initialization of the model parameters is of utmost importance to the performance of the network. We will now briefly examine some pretraining techniques which aim at properly initializing a DNN to facilitate training and improve performance.

Generative pretraining

The idea of generative pretraining stems from the fact that generative models are able to identify more complex relationships between data and target labels than discriminative models. Before describing this pretraining method it makes sense to introduce the generative models that it is based on.

Energy-based models[Ben09] Energy-based models are models basing their learning procedure on the idea of *energy*. Energy is a scalar function of each configuration of the variables of interest in our model. Such models learn by modifying the shape of the energy function so that desirable configurations of their variables have low energy.

The probability distribution of energy-based models assumes the following form:

$$P(x) = \frac{e^{-Energy(x)}}{Z}$$

where Z, called the *partition function*, is

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

If the energy function is a sum of functions f_i

$$Energy(x) = \sum_i f_i(x)$$

then the probability distribution becomes

$$P(x) \propto \prod_i P_i(x) \prod_i e^{-f_i(x)}$$

In that case, if we assume that f_i is an "expert" imposing a constraint on x, then $P(x)$ is a "product of experts" and all f_i form a distributed representation of the configuration space, where more than one expert can contribute to each region in space. In contrast, if a different energy form leads to a weighted sum of experts, i.e. $P(x)$ is a "mixture of experts", then the experts do not create a distributed representation of the configuration space, since belonging to a region excludes all the rest.

Distributed representations of data, are internal representations of the observed data in such away that they are modeled as being explained by the interactions of many latent factors ([DY14]). Such representations provide robustness in representing the internal structure of data and are capable to generalize concepts and relations.

In case there are unobservable components in our data, or we want to introduce hidden components in order to enhance the expressive power of the model, we can inset a hidden part h in the energy function. The probability distribution then becomes

$$P(v,h) = \frac{e^{-Energy(v,h)}}{Z}$$

where

$$Z = \sum_{v,h} e^{-Energy(v,h)}$$

and

$$P(v) = \sum_{h} \frac{e^{-Energy(v,h)}}{Z}$$

where v is the visible component of the data. If we define the quantity of "free energy" as

$$F_e(v) = -log(\sum_{h} e^{-Energy(v,h)})$$

then the probability distribution function of the visible part of the data takes the form of an energy-based model p.d.f. without latent components:

$$P(v) = \frac{e^{-F_e(v)}}{Z}$$

and

$$Z = \sum_{v} e^{-F_e(v)}$$

Boltzmann Machines A Boltzmann Machine is a network of symmetrically connected units that make stochastic decisions about whether to be on or off ([DY14]).

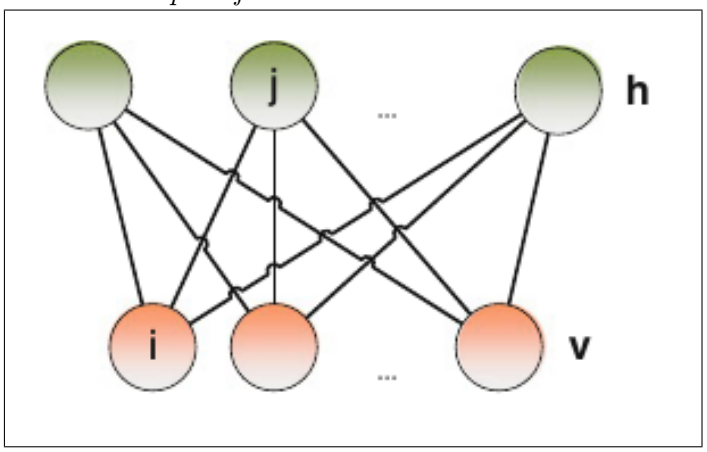
The energy function in a Boltzmann Machine is a general second-order polynomial:

$$Energy(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{a}'\boldsymbol{v} - \boldsymbol{b}'\boldsymbol{h} - \boldsymbol{h}'\boldsymbol{W}\boldsymbol{v} - \boldsymbol{v}'\boldsymbol{D}\boldsymbol{v} - \boldsymbol{h}'\boldsymbol{R}\boldsymbol{h}$$

where the offsets $\boldsymbol{a}_i, \boldsymbol{b}_i$, each associated with a single element of vector $\boldsymbol{x}$ or $\boldsymbol{h}$, and the weight matrices $\boldsymbol{W}, \boldsymbol{D}, \boldsymbol{R}$, each associated with a pair of units are the model parameters ([DY14],[Ben09]).

Restricted Boltzmann Machines A special type of Boltzmann Machine is the Restricted Boltzmann Machine (RBM). It is an undirected graphical model built by a layer of stochastic visible neurons and a layer of stochastic hidden neurons, which

Figure 1.4: *An example of a Restricted Boltzmann Machine [YD14].*



together form a bipartite graph with no visible-visible or hidden-hidden connections (figure 1.4).

The energy function of the RBM has the same form of the energy of a Boltzmann Machine, only now $\boldsymbol{D} = \boldsymbol{R} = \boldsymbol{0}$ given the special architecture of the RBM.

For the Bernoulli-Bernoulli RBM, in which $\boldsymbol{v} \in \{0,1\}^{N_v \times 1}$ and $\boldsymbol{h} \in \{0,1\}^{N_h \times 1}$ the energy is

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

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

If the visible neurons take real values, i.e. $\boldsymbol{v} \in \mathcal{R}^{N_v \times 1}$, the RBM formed is called Gaussian-Bernoulli RBM and its energy function is

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

Since RBM is an energy model, each configuration of its units $(\boldsymbol{v}, \boldsymbol{h})$ is associated with a probability

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

where

$$Z = \sum_{v,h} e^{-E(v,h)}$$

Because there are no connections between units of the same layer the posterior probabilities $P(\boldsymbol{v}|\boldsymbol{h})$ and $P(\boldsymbol{h}|\boldsymbol{v})$ can be efficiently computed. For the Bernoulli-Bernoulli RBM we can easily arrive at the expression for the posterior

$$P(\boldsymbol{h}|\boldsymbol{v}) = \prod_i \frac{e^{b_i h_i + h_i \boldsymbol{W}_{i,*}\boldsymbol{v}}}{\sum_{\bar{h}_i} e^{b_i \bar{h}_i + \bar{h}_i \boldsymbol{W}_{i,*}\boldsymbol{v}}} = \prod_i P(h_i|\boldsymbol{v})$$

where $\boldsymbol{W}_{i,*}$ is the i^{th} row of $\boldsymbol{W}$. The above expression reveals that the hidden units are conditionally independent given the visible vector. Since $h_i \in \{0, 1\}$

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

where $\sigma(\cdot)$ is the element-wise logistic sigmoid function, and for the binary visible neurons we symmetrically obtain

$$P(\boldsymbol{v} = \mathbf{1}|\boldsymbol{h}) = \sigma(\boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{a})$$

In the case of the Gaussian-Bernoulli RBM, the $P(\boldsymbol{h} = \mathbf{1}|\boldsymbol{v})$ is the same as above and

$$P(\boldsymbol{v}|\boldsymbol{h}) = \mathcal{N}(\boldsymbol{v}; \boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{a}, I)$$

where I is the identity covariance matrix.

At this point it is important to note that the posterior probability of the hidden units given the visible, independently of the input values, has the same form as the activation function of the deep multilayer perceptron (DNN) with sigmoidal hidden units. This is where the idea of generative pretraining is based, since we can equate the inference for RBM hidden units with forward computation in a DNN and use the weights of an RBM to initialize a feed-forward DNN with sigmoidal hidden units.

Using the free energy defined earlier we can express the marginal probability $P(\boldsymbol{v})$ as

$$P(\boldsymbol{v}) = \sum_{\boldsymbol{h}} P(\boldsymbol{v}, \boldsymbol{h}) = \frac{e^{-F(\boldsymbol{v})}}{\sum_{v} e^{-F(v)}}$$

For a Gaussian-Bernoulli RBM with no hidden neurons, the marginal probability density function is

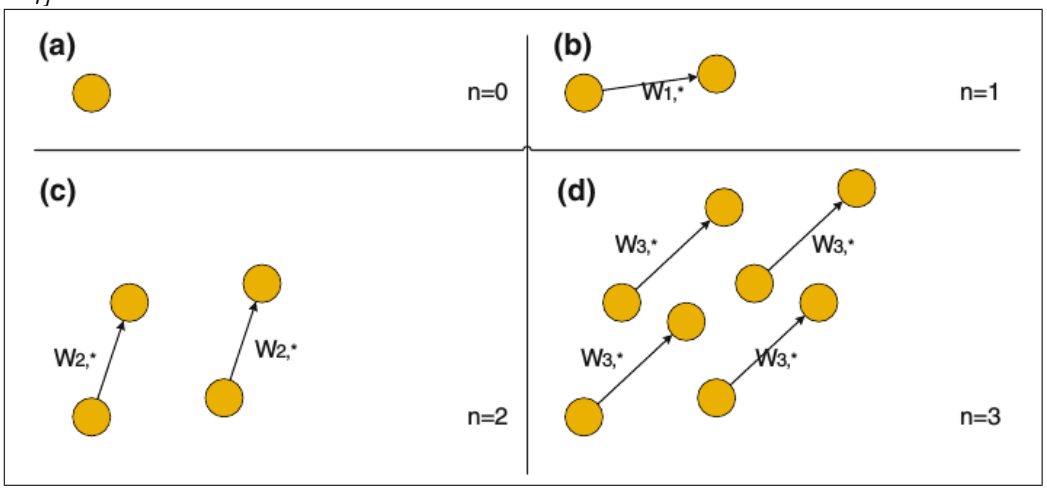
$$p_0(\boldsymbol{v}) = \frac{e^{-\frac{1}{2}(\boldsymbol{v}-\boldsymbol{a})^T(\boldsymbol{v}-\boldsymbol{a})}}{Z_0}$$

which is a unit-variance Gaussian distribution centered at vector $\boldsymbol{a}$ (figure 1.5, part (a)).

It is easy to show ([YD14]) that when a new hidden neuron is added and the rest of the model parameters are fixed, the initial distribution is scaled and a copy of it is placed along the direction determined by $\boldsymbol{W}_{n,*}$ (figure 1.5, parts (b)-(d)). This means that RBMs represent their visible inputs as a Gaussian Mixture Model with an exponential number of unit-variance Gaussians. Consequently, RBMs can be used in generative models replacing GMMs, since, for example, a Gaussian-Bernoulli RBM can represent real-valued data distributions in a similar way to GMMs.

Another useful property of RBMs is that if we present the training data to the network, it can learn the correlation between different dimensions of the feature

Figure 1.5: *Marginal probability density function of a Gaussian-Bernoulli RBM [YD14].*



vectors, i.e. represent inter-visible connections though the hidden neurons they connect to. However, this also implies that the hidden layers can be used to learn a different representation of the raw input.

When it comes to learning the RBM parameters, one can perform stochastic gradient descent using as training criterion the minimization of the negative log-likelihood:

$$J_{NLL}(\boldsymbol{W}, \boldsymbol{a}, \boldsymbol{b}; \boldsymbol{v}) = -logP(\boldsymbol{v}) = F(\boldsymbol{v}) + log \sum_{v} e^{F(v)}$$

However, the computation of the gradient of the log-likelihood of the data is infeasible to compute exactly:

$$\nabla_\theta J_{NLL}(\boldsymbol{W}, \boldsymbol{a}, \boldsymbol{b}; \boldsymbol{v}) = -\Big[\Big\langle \frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial \theta} \Big\rangle_{data} - \Big\langle \frac{\partial E(\boldsymbol{v}, \boldsymbol{h})}{\partial \theta} \Big\rangle_{model}\Big]$$

where θ is some model parameter and $\langle\cdot\rangle_{data}$ and $\langle\cdot\rangle_{model}$ are the expectation of the argument estimated from the training data and the model respectively. The first expectation $\langle\cdot\rangle_{data}$ can be computed from the training set, but $\langle\cdot\rangle_{model}$ takes exponential time to compute exactly when the hidden values are unknown. Consequently, we have to use approximated methods for RBM training, with the most widely used being the Contrastive Divergence learning algorithm ([Hin02]).

The Contrastive Divergence algorithm aims to locally approximate the gradient of the training criterion around a training point. Its goal is to discover a decision surface to separate high- from low-probability regions by comparing training samples and samples generated by the model. The term "contrastive" comes exactly from the fact that it builds on the contrast between these two classes of samples.

The algorithm starts by initializing a sampling process (*Gibbs sampling*) at a training data sample. It then generates a hidden sample from the visible sample

Figure 1.6: *Contrastive Divergence algorithm for RBM training [YD14].*

procedure TRAINRBMWITHCD($\mathbb{S} = \{\mathbf{o}^m | 0 \leq m < M\}, N$)
▷ $\mathbb{S}$ is the training set with M samples, N is the CD steps
  Randomly initialize $\{\mathbf{W_0}, \mathbf{a}_0, \mathbf{b}_0\}$
  while Stopping Criterion Not Met **do**
    ▷ Stop if reached max iterations or the training criterion improvement is small
    Randomly select a minibatch $\mathbf{O}$ with M_b samples.
    $\mathbf{V}^0 \leftarrow \mathbf{O}$ ▷ Positive phase
    $\mathbf{H}^0 \leftarrow P(\mathbf{H}|\mathbf{V^0})$ ▷ Applied column-wise
    $\nabla_{\mathbf{W}} J \leftarrow \mathbf{H^0} \left(\mathbf{V^0}\right)^{\mathrm{T}}$
    $\nabla_{\mathbf{a}} J \leftarrow \text{sumrow}\left(\mathbf{V}^0\right)$ ▷ Sum along rows
    $\nabla_{\mathbf{b}} J \leftarrow \text{sumrow}\left(\mathbf{H^0}\right)$
    for $n \leftarrow 0; n < N; n \leftarrow n + 1$ **do** ▷ Negative phase
      $\mathbf{H}^n \leftarrow \mathbb{I}\left(\mathbf{H}^n > \text{rand}\,(0, 1)\right)$ ▷ Sampling, $\mathbb{I}(\bullet)$ is the indicator function
      $\mathbf{V}^{n+1} \leftarrow P(\mathbf{V}|\mathbf{H}^n)$
      $\mathbf{H}^{n+1} \leftarrow P(\mathbf{H}|\mathbf{V}^{n+1})$
    end for
    $\nabla_{\mathbf{W}} J \leftarrow \nabla_{\mathbf{W}} J - \mathbf{H}^N \left(\mathbf{V}^N\right)^{\mathrm{T}}$ ▷ Subtract negative statistics
    $\nabla_{\mathbf{a}} J \leftarrow \nabla_{\mathbf{a}} J - \text{sumrow}\left(\mathbf{V}^0\right)$
    $\nabla_{\mathbf{b}} J \leftarrow \nabla_{\mathbf{b}} J - \text{sumrow}\left(\mathbf{H^0}\right)$
    $\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t + \frac{\varepsilon}{M_b} \triangle \mathbf{W}_t$ ▷ Update $\mathbf{W}$
    $\mathbf{a}_{t+1} \leftarrow \mathbf{a}_t + \frac{\varepsilon}{M_b} \triangle \mathbf{a}_t$ ▷ Update $\mathbf{a}$
    $\mathbf{b}_{t+1} \leftarrow \mathbf{b}_t + \frac{\varepsilon}{M_b} \triangle \mathbf{b}_t$ ▷ Update $\mathbf{b}$
  end while
  Return $rbm = \{\mathbf{W}, \mathbf{a}, \mathbf{b}\}$
end procedure

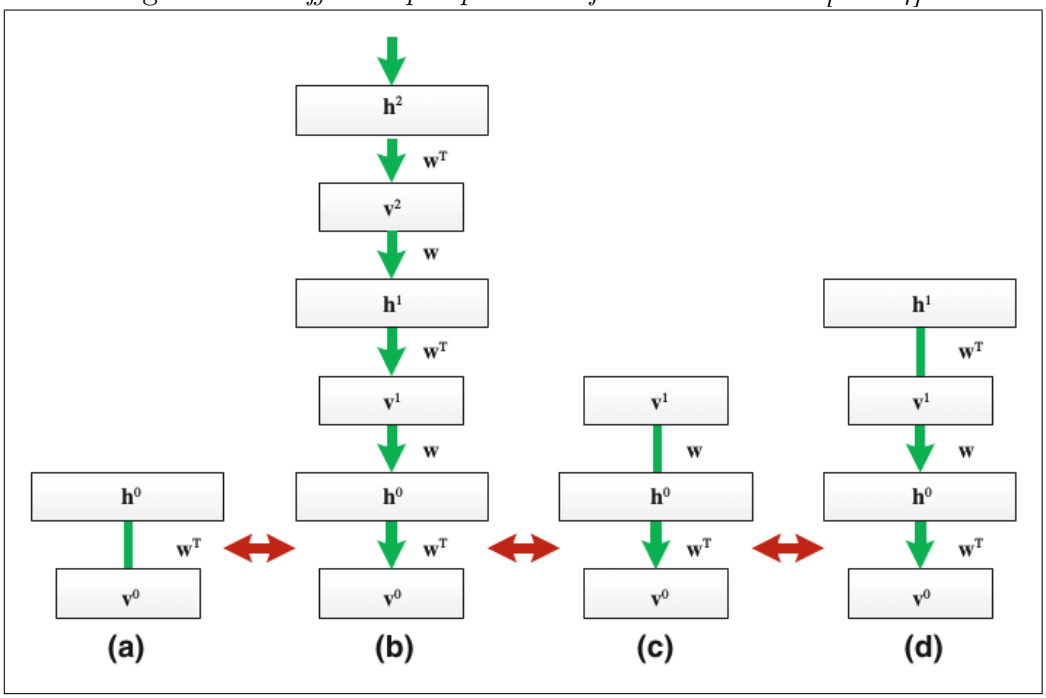
based on the posterior probability $P(\boldsymbol{h}|\boldsymbol{v})$, defined accordingly to the RBM model used (Gaussian-Bernoulli or Bernoulli-Bernoulli). This hidden sample is further used to generate a visible sample based on the posterior probability $P(\boldsymbol{v}|\boldsymbol{h})$. This process may continue for many steps. If it continues for an infinite number of steps the true expectation $\langle\cdot\rangle_{model}$ can be estimated. However, it has been found that even one step of the algorithm is enough to provide us with a good estimate of $\langle\cdot\rangle_{model}$ and consequently of the gradient of the training criterion ([Hin12]).

Beep Belief Networks We can view an RBM as a generative model with infinite number of layers all of which share the same weight matrix (figure 1.7, parts (a)-(b)).

If we separate the bottom layer from the deep model of figure 1.7(b) the remaining layers form another generative model with infinite number of layers sharing the same weight matrices, i.e. they are equivalent to another RBM whose visible layer and hidden layer are switched (figure 1.7, part (c)). This model is a special generative model called Deep Belief Network (DBN), where the bottom layers form a directed generative model and the top layer is an undirected RBM. Thinking in the same way we can derive the equivalent model shown in figure 1.7(d).

Because the DBN is related closely to the RBM, we can apply a layer-wise process

Figure 1.7: *Different perspectives of the same RBM [YD14].*



to train deep generative models. We start by training an RBM, which, after the training, can discover a different representation of the feature vectors. Hence, for each vector $\boldsymbol{v}$ we compute a vector of expected hidden neuron activations $\boldsymbol{h}$. These hidden expectations can be fed to a new RBM as training data. Continuing in this way we can use each set of RBM weights to extract features from the output of the previous layer. When we do not need to train more RBMs, we can use the weight matrices to initialize the hidden layers of a DBN with as many hidden layers as the RBMs we have trained. The final DBN can be further fine-tuned.

This procedure can be used to stack RBMs with different or the same number of dimensions which allows us to improve the flexibility and performance of the DBN.

Weight initialization of a DNN The DBN weights can be used to initialize the weights of a sigmoidal DNN. This is due to the fact that $P(\boldsymbol{h}|\boldsymbol{v})$ in the RBM has the same form as that in the DNN provided the layers use the sigmoidal activation function. The DNN can be viewed as a statistical graphical model where each hidden layer $0 < l < L$ models posterior probabilities of conditionally independent hidden binary neurons $\boldsymbol{h}^l$ given input vectors $\boldsymbol{v}^{l-1}$ as Bernoulli distribution

$$P(\boldsymbol{h}^l|\boldsymbol{v}^{l-1}) = \sigma(\boldsymbol{z}^l) = \sigma(\boldsymbol{W}^l\boldsymbol{v}^{l-1} + \boldsymbol{b}^l)$$

and the output layer approximates the label $\boldsymbol{y}$ as

$$P(\boldsymbol{y}|\boldsymbol{v}^{L-1}) = softmax(\boldsymbol{z}^L) = softmax(\boldsymbol{W}^L\boldsymbol{v}^{L-1} + \boldsymbol{b}^L)$$

Given the observation vector $\boldsymbol{o}$ and its label $\boldsymbol{y}$, the precise modeling of $P(\boldsymbol{y}|\boldsymbol{o})$ requires integration over all possible values of $\boldsymbol{h}$ across all layers. However, because this is infeasible, we replace the marginalization with the mean-field approximation:

$$\boldsymbol{v}^l = \mathrm{E}(\boldsymbol{h}^l|\boldsymbol{v}^{l-1}) = P(\boldsymbol{h}^l|\boldsymbol{v}^{l-1}) = \sigma(\boldsymbol{W}^l\boldsymbol{v}^{l-1} + \boldsymbol{b}^l)$$

which is the non-stochastic description of the DNN previously presented.

Weight initialization via generative pretraining can improve the DNN performance on the test set. This is due to three main reasons. First, the initial point of the learning algorithm can greatly affect the end model, especially in batch training. Second, since we only need labels for the fine-tuning part of the training, we can utilize a large amount of unlabeled data, which would be useless without pretraining. Third, the generative criterion used in pretraining is different from the discriminative criterion used in fine-tuning, which means that pretraining can act as a data-dependent regularizer.

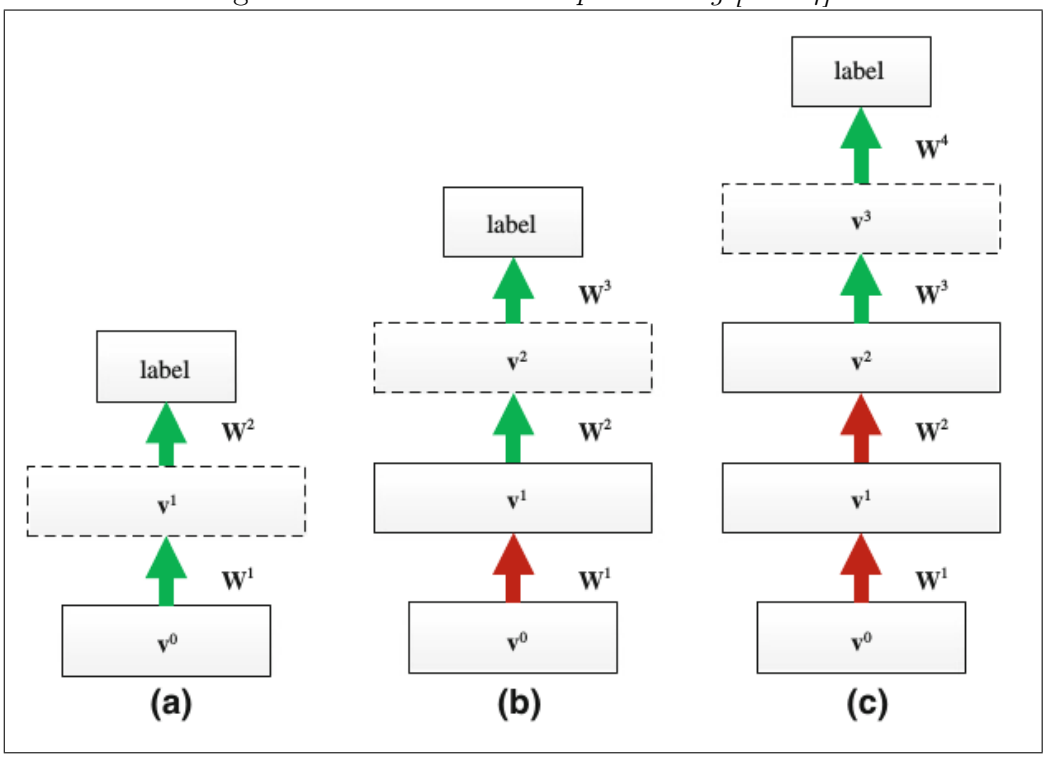
Bengio in [Ben09] debates whether unsupervised (generative) pretraining has in principle a regularization or optimization effect.

Given that generative pretraining is equivalent to imposing a constraint on where in the parameter space a solution is allowed, i.e. near solutions of the unsupervised training criterion that capture important statistical structure in the data, it can be seen as a form of regularization. Furthermore, generative pretraining plays a more important role in the initialization of the lower layers. Experiments have shown that when there are no constraints on the number of hidden neurons at the top two layers of the network, the representation computed by the lower layers is unimportant: the network can still be trained to minimize the training error as much as desired - despite the fact that it may generalize poorly. However, if we have to use smaller top layers, then pretraining of the lower layers becomes necessary to get a low training error and better generalization.

The optimization perspective of unsupervised pretraining requires to focus on tuning the lower layers while the top two layers are kept small in terms of the number of neurons or of the magnitude of their weights. Bengio suggests that if unsupervised pretraining worked purely as a regularizer, then if we had an infinite stream of training data and performed online learning - in which case we would in essence minimize the generalization error - then with or without pretraining the error would converge to the same level. However, after emulating such a setting, he discovered that the pretrained network achieved a lower minimum, which suggests that there is an optimization component in generative pretraining.

In any case, there has been no evidence so far of generative pretraining hurting the DNN training.

Figure 1.8: *Discriminative pretraining [YD14].*



Apparently, generative pretraining is highly dependent on the task. It also works best with two hidden layers and is of trivial importance when the network has only one hidden layer. If we add more hidden layers effectiveness often decreases, since modeling errors introduced by the mean-field approximation and the contrastive divergence algorithm accumulate ([YD14]).

Discriminative pretraining

A different, "incremental", approach to pretraining is to discriminatively train each layer using back-propagation. Pretraining begins by training a one-hidden-layer network to convergence, using labels. Then, we insert another hidden layer before the output layer, randomly initialized, and retrain the whole network to convergence. We continue in the same way until the network reaches the desired number of hidden layers (figure 1.8)

It is important to note that all hidden layers are updated during training and not only the layer added last. Because of this, if we are planning to add more hidden layers, the network should not be trained to full convergence, to avoid having some hidden neurons operating in the saturation region and thus being unable to further update. A common heuristic is to go through the training set $\frac{1}{L}$ times of the total number of the data passes needed to convergence, where L is total number of layers

in the final model.

Discriminative pretraining aims to bring the weights close to a good local optimum and it does not have the regularization effect of generative pretraining. Therefore, it is best used when large amounts of training data are available.

Hybrid pretraining

Both pretraining techniques presented work well, however they both have some drawbacks.

Generative pretraining is not directly related to the task-specific objective function, hence it is not guaranteed to help the discriminative fine-tuning phase. However, it does help to reduce overfitting. On the other hand, discriminative pretraining minimizes the training criterion, yet it entails the danger of over-tuning the lower layers to the final objective thus making them unable to learn when new hidden layer are added.

To tackle these issues, it has been suggested to optimize a weighed sum of the generative and discriminative pretraining criteria. We thus have a hybrid criterion to optimize:

$$J_{HYB}(\boldsymbol{W}, \boldsymbol{b}; \mathcal{D}_{train}) = J_{DISC}(\boldsymbol{W}, \boldsymbol{b}; \mathcal{D}_{train}) + \alpha J_{GEN}(\boldsymbol{W}, \boldsymbol{b}; \mathcal{D}_{train})$$

The discriminative criterion can be cross-entropy or mean-square error whereas the generative can be negative log-likelihood. Intuitively, the generative criterion acts as a data-dependent regularizer for the discriminative.

Hybrid pretraining can outperform both the generative and the discriminative pretraining approaches. Despite the fact the the importance of pretraining diminishes as the amount of training data increases, pretraining can still help to make the training procedure more robust.

Dropout pretraining

We have already seen that dropout can be considered a bagging technique, thus generate a smoother objective surface. This implies that dropout could be used to pretrain a DNN to easily find a good starting point on the smoothed objective surface and then fine-tune the model without dropout.

Dropout requires labeled data and achieves similar performance to the discriminative pretraining, as well as being easier to implement and control.

1.4 Recurrent Neural Networks

[YD14]

Research in ASR has been working towards building models that can adequately emulate the human speech production and perception system. However, as we have seen, GMM/HMM models fail to model the dynamic and hierarchical structure of the human speech system and DNN/HMM models exhibit the same type of limitations. A Recurrent Neural Network (*RNN*) is a class of neural network models where many connections among its neurons form a directed cycle (hence the name *recurrent*), thus providing the network with a structure of internal states, or memory, which helps it exhibit a dynamic temporal behavior missing from the DNN.

The internal representation of dynamic speech features in the RNN is discriminatively formed by feeding the low-level features into the hidden layers, together with the hidden features from the past history. The RNN implements a time-delay operation over the temporal dimension which allows its internal states to function as memory and the network to exhibit a dynamic temporal behavior.

RNNs are considered deep models since if we unfold the network in time, we will create as many layers in the network as the length of the input speech utterance.

In this section we will present the fundamental RNN model and the two basic ways to train it, as well as a model which has received a lot of attention recently, the *Long-Short-T Term Memory*.

1.4.1 State-Space formulation of the basic RNN

The RNN differs from the DNN in that it operates not only on its inputs but also on its internal states, which encode the past information that has already been processed.

We will express the one-hidden layer RNN in terms of the noise-free nonlinear state-space model often used in signal processing, which will enable the learning of sequentially extended dependencies over a long time span. If $\boldsymbol{x}_t$ is the $K \times 1$ input vector, $\boldsymbol{h}_t$ is the $N \times 1$ vector of hidden state values and $\boldsymbol{y}_t$ is the $L \times 1$ output vector, the one-hidden layer RNN can be described as

$$\boldsymbol{h}_t = f(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1}) \ (state\ equation)$$

$$\boldsymbol{y}_t = g(\boldsymbol{W}_{hy}\boldsymbol{h}_t) \ (observation\ equation)$$

where $\boldsymbol{W}_{hy}$ is the $L \times N$ weight matrix connecting the N hidden units to the L outputs, $\boldsymbol{W}_{xh}$ is the $N \times K$ weight matrix connecting the K inputs to the N hidden units, and $\boldsymbol{W}_{hh}$ is the $N \times N$ weight matrix connecting the N hidden units from time $t-1$ to time t.

Let us further define

$$\boldsymbol{u}_t = \boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1}$$

as the $N \times 1$ vector of hidden layer potentials and

$$\boldsymbol{v}_t = \boldsymbol{W}_{hy}\boldsymbol{h}_t$$

as the $L \times 1$ vector of output layer potentials.

Then, $f(\boldsymbol{u}_t)$ is the hidden layer activation function and $g(\boldsymbol{v}_t)$ is the output layer activation function. Commonly used activation functions in the hidden layers are the *sigmoid*, *tanh* and rectified linear units ($ReLU$), whereas typical output layer activation functions are the *linear* and $softmax$ functions.

In case we want to use the output from previous time frames to update the state vector the state equation becomes

$$\boldsymbol{h}_t = f(\boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{yh}\boldsymbol{y}_{t-1})$$

where $\boldsymbol{W}_{yh}$ is the weight matrix connecting the output layer to the hidden layer.

1.4.2 Training

Back-propagation-through-time algorithm

The back-propagation-through-time ($BPTT$) method ([Bod01]) manages to learn the weight matrices of the RNN by first unfolding the network in time and then propagating errors backwards through time. The algorithm is an extension of the original back-propagation algorithm for feed-forward neural networks, yet now the stacked hidden layers for the same training frame t have been replaced by the T same single hidden layers across time $t = 1, 2, ..., T$.

The training criterion of the BPTT algorithm is the well-known sum-square error

$$E = \frac{1}{2}\sum_{t=1}^{T} \|\boldsymbol{l}_t - \boldsymbol{y}_t\|^2$$

where $\boldsymbol{l}_t$ is the target vector and $\boldsymbol{y}_t$ is the output vector over all time frames. Our goal is to minimize this cost with respect to the weights and for that we will use the gradient descent rule. The algorithm is presented in figure 1.9

It should be noted that contrary to the DNN backpropagation algorithm, the RNN weight matrices are spatially duplicated for an arbitrary number of time steps, i.e. they are *tied*. That is the reason why in the update step of the algorithm we sum over all time frames.

The computational complexity of the BPTT is $O(M^2)$ per time step, where $M = LN + NK + N^2$ is the total number of the model parameters. It converges slower the the original back-propagation algorithm due to dependencies between frames, and is highly likely to converge to a poor local optimum due to exploding or

Figure 1.9: *Backpropagation-through-time [YD14].*

procedure BPTT($\{\mathbf{x}_t, \mathbf{I_t}\}\, 1 \leq t \leq T$)
▷ $\mathbf{x}_t$ is the input feature sequence
▷ $\mathbf{I_t}$ is the label sequence
▷ forward computation
  for $t \leftarrow 1; t \leq T; t \leftarrow t+1$ **do**
    $\mathbf{u}_t \leftarrow \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1}$
    $\mathbf{h}_t \leftarrow f(\mathbf{u}_t)$
    $\mathbf{v}_t \leftarrow \mathbf{W}_{hy}\mathbf{h}_t$
    $\mathbf{y}_t \leftarrow g(\mathbf{v}_t)$
  end for
▷ backpropagation through time
  $\delta_T^y \leftarrow (\mathbf{l}_T - \mathbf{y}_T) \bullet g'(\mathbf{v}_T)$ ▷ •: element-wise multiplication
  $\delta_T^h \leftarrow \mathbf{W}_{hy}^T \delta_T^y \bullet f'(\mathbf{u}_T)$
  for $t \leftarrow T-1; t \geq 1T; t \leftarrow t-1$ **do**
    $\delta_t^y \leftarrow (\mathbf{l}_t - \mathbf{y}_t) \bullet g'(\mathbf{v}_t)$
    $\delta_t^h \leftarrow \left[\mathbf{W}_{hh}^T \delta_{t+1}^h + \mathbf{W}_{hy}^T \delta_t^y\right] \bullet f'(\mathbf{u}_t)$ ▷ propagate from δ_t^y and δ_{t+1}^h
  end for
▷ model update
  $\mathbf{W}_{hy} \leftarrow \mathbf{W}_{hy} + \gamma \sum_{t=1}^{T} \delta_y^t \mathbf{h}_t^T$
  $\mathbf{W}_{hh} \leftarrow \mathbf{W}_{hh} + \gamma \sum_{t=1}^{T} \delta_h^t \mathbf{h}_{t-1}^T$
end procedure

vanishing gradients. Although we can limit the past history to the last p time steps and thus improve speed, it is still difficult to achieve good results without much experimentation and tuning.

Bibliography

[Ben09] Yoshua Bengio. Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1):1–127, January 2009.

[Ben12] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533, 2012.

[Bod01] Mikael Boden. A guide to recurrent neural networks and backpropagation, 2001.

[DY14] Li Deng and Dong Yu. Deep learning: Methods and applications. *Found. Trends Signal Process.*, 7(3–4):197–387, June 2014.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*, 2010.

[Hin02] Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Comput.*, 14(8):1771–1800, August 2002.

[Hin12] Geoffrey E. Hinton. A practical guide to training restricted boltzmann machines. In Gregoire Montavon, Genevieve B. Orr, and Klaus-Robert MG'Oller, editors, *Neural Networks: Tricks of the Trade (2nd ed.)*, volume 7700 of *Lecture Notes in Computer Science*, pages 599–619. Springer, 2012.

[HS52] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49:409–436, 1952.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680, 1983.

[LBOM98] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Effiicient backprop. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 9–50, London, UK, UK, 1998. Springer-Verlag.

[LN89] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Math. Program.*, 45(3):503–528, December 1989.

[Nie15] Michael Nielsen. Neural Networks and deep learning,. , 2015.

[YD14] Dong Yu and Li Deng. *Automatic Speech Recognition: A Deep Learning Approach*. Springer Publishing Company, Incorporated, 2014.