Training HMMs in GMTK

John T. Halloran
Department of Electrical Engineering
University of Washington

June 15, 2015

Abstract

The Graphical Models ToolKit (GMTK) is a powerful and flexible prototyping language for designing dynamic Bayesian networks (DBNs). This tutorial is meant to help new users' understanding of GMTK by presenting hidden Markov models (HMMs) which make use of some of the software's large number of features. Generative and discriminative training approaches supported in GMTK are discussed with relevant examples, as well as testing using an HMM for a simple classification task. All described models and scripts are available in the tarball housing this document.

The following examples are intended for those interested in using the Graphical Models ToolKit (GMTK) to perform various forms of inference/learning utilizing graphical models. Using GMTK, we will first train and test a *hidden Markov model* (HMM) whose emission states are discrete (Section 1), then train and test one whose emission states are real valued (conditionally Gaussian) (Section 2). Also discussed are generative and discriminative training approaches supported in GMTK.

1 Classifying the weather: discrete observations

Consider the scenario wherein the area you work/reside only encounters three types of weather: sunny, rainy, and foggy. Assume that you go to work in an office everyday, and during your working hours you do not get to see nor experience a given days weather (you're office also has no windows looking outside; this was my office in Hawaii ironically enough). Curiosity mounts as time goes on and you become more and more curious about what the daily weather is. Your hope lies in that you have an office mate who comes in hours later than you, and this office mate brings in an umbrella some days, and no umbrella all other days.

In the framework of the HMM, the hidden layer is the state of the weather (indeed, it is hidden from you). The observed layer consists of the daily observations of weather your office mate has brought

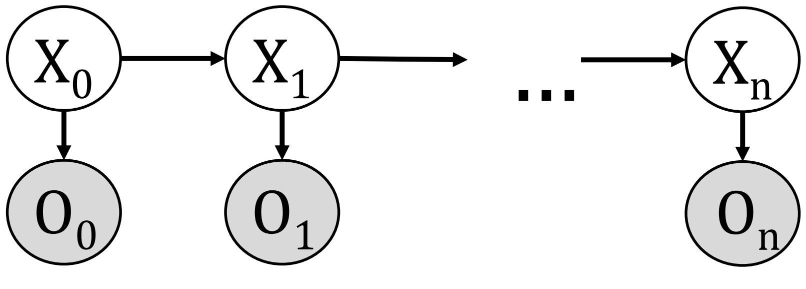


Figure 1: Hidden Markov Model (HMM) of length n. Shaded nodes, O_t are observed, unshaded, X_t are hidden. The Markov chain $X_{t-1} \to X_t \to X_{t+1}$ is called the hidden layer, while the lower level of nodes is called the observed layer.

an umbrella to the office or not. Formally, let X be a random variable such that $X \in \{0, 1, 2\}$, where state 0 denotes sunny, state 1 denotes rainy, and state 2 denotes foggy. We assume that the Xs are first-order Markov, i.e., denoting a specific day by t and the corresponding random variable describing the weather of that day as X_t, X_{t+1} is indepedent of X_{t-1} given X_t. Let O be a random variable s.t. $O \in \{0, 1\}$, where $O_t = 0$ if your coworker did not bring an umbrella into the office that day and $O_t = 1$ otherwise. The graphical representation of the HMM is available in figure 1.

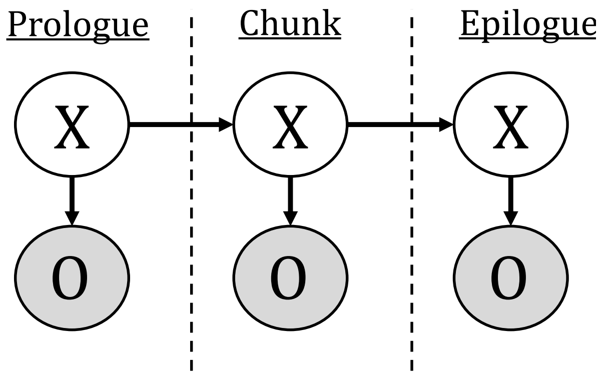
In the graph, O_t is a child of X_t, as is X_{t+1} for $t > 0$ (assume that all observations began at day 0). Thus, after n days, the joint distribution over these variables will be the following:

$$\begin{aligned}
p(X_0, X_1, X_2, \ldots, X_n, O_0, \ldots, O_n) &= p(X_0)p(X_1|X_0)p(X_2|X_0, X_1)\ldots \\
&\quad p(O_0|X_0, \ldots, X_n)p(O_1|O_0, X_0, \ldots, X_n)\ldots \\
&\quad p(O_n|O_0, \ldots, O_{n-1}, X_0, \ldots, X_n), \quad \text{by Bayes' rule} \\
&= p(X_0)p(X_1|X_0)p(X_2|X_1)\ldots \\
&\quad p(O_0|X_0)p(O_1|X_1)p(O_n|X_n) \\
&\quad \text{by the HMM conditional independence properties} \\
&= p(X_0)\prod_{t=1}^{n} p(X_t|X_t - 1)p(O_t|X_t)
\end{aligned}$$

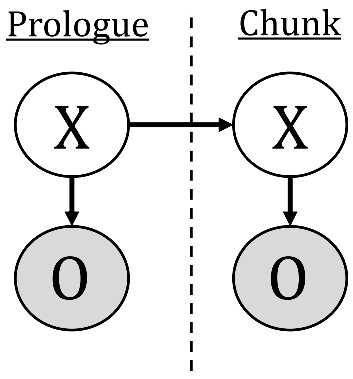
1.1 DBNs and GMTK: training/testing this HMM

In the context of a DBN, the graph is defined arbitrarily over a sequence of length n wherein each time (day in the weather example) instance is called a frame. When n is instantiated, i.e., we oberve a particular sequence, the DBN is arbitrarily unrolled to fill all instances of the n sequence samples. In our case, the DBN turns out to be an HMM. The prerequisite definitions for this graph being defined for an arbitrary sequence are a graph which defines the first sample of any given sequence ($t = 0$), and a arbitrary graph for $t > 0$.

In GMTK speak, the first frame is called the prologue, and defines what happens over the first observed sequence sample. The following frame is called the chunk, and this is the workhorse of the DBN. Essentially, the chunk is specified to unroll for the following $0 < t < n$ sequence samples. Finally, the last frame is called the epilogue, and this details what happens in the final sample. One can also imagine that the prologue, chunk, and epilogue can be defined for multiple frames, and this would just amount to describing the dependencies over runs of your observed sequence. The semantics will remain the same, in that the chunk is specified to unroll and essentially fill the middle portion of a sequence. The prologue, chunk, and epilogue of the HMM are shown below. Note that the epilogue is just a replica of the chunk. There is no restriction which says that the epilogue need to be defined, and if you can recursively define the graph you are interested in without this you are free to do so in GMTK.



(a) HMM template in GMTK



(b) Equivalent HMM templates in GMTK

Figure 2: HMM templates in GMTK. The two are equivalent since in figure 2a, the chunk is the same as the epilogue, so the chunk need only be unrolled over the last frame as is the case for the template in figure 2b.

1.1.1 Graph structure definition

To define the structure of the graph in GMTK, we will create a *structure file*, `hmm.str`. The objects described in this file will be the random variables of the graph. To describe the variables in the prologue of the HMM in figure 2, the following is the gmtk syntax:

```
frame: 0 {
variable : X {
type : discrete hidden cardinality 3;
conditionalparents : nil using DenseCPT("pX");
  }

variable : O {
type : discrete observed 0:0 cardinality 2;
conditionalparents : X(0) using DenseCPT("pO_given_X");
  }

}
```

The frame for which we are describing variables is first defined, frame 0. Within this frame we have two variables, X and O. There are 3 types of random variables in GMTK: discrete and hidden (i.e., random in which case we iterate over all values of the random variable during inference), discrete and observed (these variables do not vary during inference, and we could further define them to be deterministic or define a pdf which assigns a probability to their observed value), and real valued and observed (described in Section 2). All hidden variables have cardinality C and it is assumed that they take on values in the range $[0, C-1]$. In frame 0, $X(0)$ is discrete and hidden. Since this is the first frame, $X(0)$ has no parents, which is designated by conditonalparents : nil. The marginal distribution over X is a DenseCPT called pX, the symantics of which will be described in the next section.

$O(0)$ is discrete and observed, as evidenced by the type declaration. The value that $O(0)$ takes on is fed into GMTK via what is called the *global observation matrix*. Basically, a file containing the sequence of data is specified on the command line and loaded into memory. The syntax $0:0$ designates a range in the global observation matrix. Corresponding to our binary sequence (umbrella observations), the global observation matrix will consist of a single column, and each row will correspond to each frame in the model. So, if we had a sequence 010, then the global observation matrix would be $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$, the range specification $0:0$ would associate each observation to the first element of each row, ie $O(0) = 0$, $O(1) = 1$, and $O(2) = 0$. Finally, $O(0)$ has a single parent $X(0)$, the conditional probability table (DenseCPT) of which is called pO_given_x. One important thing to note is that the index of a random variable in a frame is relative only to its local frame value; the designation $X(0)$ says the value of random variable X in the current frame, whereas $X(-1)$ would say the value of the random variable X is the previous frame. This will become more obvious when describing the next frame (the chunk) of the HMM.

To complete the HMM, we describe the chunk:

```
frame: 1 {
variable : X {
type : discrete hidden cardinality 3;
conditionalparents : X(-1) using DenseCPT("pX_given_X");
  }
variable : O {
type : discrete observed 0:0 cardinality 2;
conditionalparents : X(0) using DenseCPT("pO_given_X");
  }

}

chunk 1:1;
```

Many of the semantics are the same as we have seen for the prologue; we define random variables associated with a particular frame and describe their conditional dependence relationships. The line chunk 1 : 1 says the range of frames which are used for the chunk. Since we only have frame 1 which unrolls for the $n-1$ last data sequences, 1 : 1 says to use frame 1 to arbitrarily unroll upon all but the first observed data sequence. As previously mentioned, the designation of $X(-1)$ as a conditionalparent of X in this frame means that after the chunk has been unrolled, X_0 will be a parent of X_1 corresponding to the prologue and the first unrolled chunk frame, X_1 will be a parent of X_2 corresponding to the first and second unrolled chunk frames, and so on.

1.1.2 Trained parameter files: defining CPTs

There are several ways of representing probability distributions in GMTK. Real valued conditional distributions will be covered in the sequel. The most natural way of specifying a discrete distibution is via a table/matrix of probability values. This is called a *DenseCPT* in GMTK, the syntax of which is as follows

```
number of DenseCPTs to follow

%notation for a CPT:
cpt_index
cpt_name
number of parents
cardinality of parents
cardinality of self
dense_cpt values
```

where % denotes a comment. First, we define the number of DenseCPTs to follow, say N. Moving on to define a particular CPT, we begin by referring to that CPTs index (its in-file rank starting from 0 and ending at $N-1$). Next we define the name of the CPT, followed by the number of parents of the random variable the CPT describes, followed by the cardinality of the random variable's parents, followed by the cardinality of the random variable itself, and finally the matrix of probability values. For instance, the following would be the definition of all the HMM CPTs set to uniform probability distributions.

```
% Dense CPTs
3
0
pX
0 % number parents
3 % cardinalities
```

```
0.33333333333 0.33333333333 0.33333333333

1
pX_given_X
1 % number parents
3 3 % cardinalities
0.33333333333 0.33333333333 0.33333333333
0.33333333333 0.33333333333 0.33333333333
0.33333333333 0.33333333333 0.33333333333

2
pO_given_X
1 % number parents
3 2 % cardinalities
0.5 0.5
0.5 0.5
0.5 0.5
```

1.2 GMTK training and testing

Two bash scripts, train_cmd.sh and test_cmd.sh, are included in the tarball and should be runnable upon untarring. To run these, make sure the gmkt binary (or at least gmtkEMtrain and gmtkViterbi) are in your path.

In order to infer the most probable assignment of the hidden layer to explain the observed layer (called Viterbi decoding, wherein we calculate the configuration of hidden states which maximizes the joint distribution, amongst all possible hidden configurations), we would first like to train our data. GMTK features maximum estimation to learn both multinomial and Gaussian parameters via the *expectation-maximization* (EM) algorithm [1]. To proceed with training our multinomial distributions, one can specify a file of initial parameters for the distributions. In the accompanying tarball, this file is `init_hmm.params` and follows the semantics of the section describing CPTs. Next, the binary gmtkEMtrain is called with various switches and parameters. To take a look at this call, please see train_cmd.sh. The switches and their definitions are:

strFile	the the structure file
inputMasterFile	the master file (empty for this example)
inputTrainableParameters	file of initial CPT values
outputTrainableParameters	file to write output CPT values
of1	observed file 1, file containing global list of sequences (one file per sequence in ascii)
fmt1	format for observed file 1, ascii in our case
nf1	number of floats in observed file 1 (0)
ni1	number of ints in observed file 1 (1, umbrella/no umbrella)
dirichletPriors	boolean, use or don't use dirichlet priors (T is true)
maxE	maximum number of EM iterations
lldp	threshold at which to stop based on consecutive values between iterations
objsNotToTrain	file listing CPTs not to train
random	boolen, randomly initialize all values (T is true)

Once training completes, we're ready to calculate the Viterbi assignment To do so, we'll be using the binary gmtkViterbi. The bash script which calls this binary is test_cmd.sh. Many of the switches are the same as those for gmtkEMtrain (and gmtk binaries in general). The switches which are distinct and worth noting are:

verbosity	integer value specification ranging from 0 to 99; output to terminal, ranges from showing the log-likelihood probabilities (verb 10), to showing the instantiation of variables in message-passing (verb 66), to the probability of variables in message-passing (verb 86), and so on.
vitValsFile	file to wrie the viterbi path to

All training and testing data are contained in the tarball subdirectory data, which also contains the matlab script used to generate the data. The true labels are also included in the data directory, both for the testing and training data. Running test_cmd.sh will perform a verification of the Viterbi assignment versus the ground truth labels. One should be able to achieve around 70% accuracy with the current data generation.

2 Classifying the weather: real valued observations

As in section 1, you are curious about the weather wherein you work, which only happens to be sunny, rainy, and foggy. However, your coworker no longer brings umbrellas to work. Instead, your coworker utters a real number each day which is a noisy representative of that day's temperature. Our model of the weather now changes such that $O_t \in \mathbb{R}$. Assuming this noise to be Gaussian, we have that $p(O_t|X_t = x) = \mathcal{N}(\mu_x, \sigma_x^2)$. Once more using an HMM, the graph of the model is again that of Figure 2a. However, where previously our emission probabilities were discrete, they are now Gaussian. The pertinent change in observation variable declaration is as follows (as seen in file `hmm_gaussLeaves.str`):

```
variable : O {
type : continuous observed 0:0;
conditionalparents : X(0) using mixture collection("pO_given_X")
                     mapping("internal:copyParent");
  }
```

O declared as above for both the prologue and chunk. $O(0)$ is defined as real valued and observed (the only type of real valued/continuous variable supported in GMTK). Here, $O(0)$ has a single parent, $X(0)$. Each Gaussian variable in GMTK is defined as a mixture of Gaussians, where `mixture collection` refers to a *collection* of Gaussian mixtures (this may be thought of as an array) corresponding to $p(O(0)|X(0) = x)$. As $|X| = 3$, there must be 3 Gaussian mixtures in the collection `pO_given_X`. To make this more explicit, we'll be using single dimensional Gaussians $p(O(0)|X(0) = x) = \mathcal{N}(\mu_x, \sigma_x^2)$ so that, when $x = 1$, $p(O(0)|X(0) = 1) = \mathcal{N}(\mu_1, \sigma_1^2)$ and so on for $x \in \{2, 3\}$.

Now, define our collection of Gaussian mixtures to be the vector v with ith element $v(i)$. Each Gaussian in a mixture is referred to as a *component*, so that, for an arbitrary n-component mixture $M(\mu, \sigma^2, a)$ where $\mu = [\mu(1), \dots, \mu(n)]^T$, $\sigma^2 = [\sigma^2(1), \dots, \sigma^2(n)]^T$, and $a = [a(1), \dots, a(n)]^T$ we have $M(\mu, \sigma^2, a) = \sum_{i=1}^n a(i)\mathcal{N}(\mu(i), \sigma^2(i))$. Here, a is a vector of *mixture coefficients* (also known as component responsibilities or Gaussian occupancies) such that $\sum_{i=1}^n a(i) = 1$, i.e., a is really a distribution over the Gaussians in the mixture. For our example, we let $n = 1$, so that each mixture simply consists of a single Gaussian. Thus, a is a scalar and $a = 1$. Note, however, that GMTK supports arbitrary n values as well as learning for the Gaussian means, variances, and mixture coefficients. Our vector of Gaussian mixtures is then $v = [M(\mu_1, \sigma_1^2, 1), M(\mu_2, \sigma_2^2, 1), M(\mu_3, \sigma_3^2, 1)]^T$. Our conditional distribution (emission distribution), then, is simply $p(O(0)|X(0) = i) = v(i)$, for $i \in \{1, 2, 3\}$.

The argument to `mapping` is a *decision tree* (DT) which defines a deterministic mapping based on the parents of a random variable. The DT `internal:copyParent` defines an identity mapping such that, when $O(0)$'s parent $X(0) = i$, we choose the ith Gaussian mixture in our vector of mixtures v. Much more complicated deterministic mappings are possible, though this is outside the scope of this document. It is worth noting, however, that deterministic functions of children may be defined in GMTK using DTs; that is, for a random variable Y with parents $\pi(Y)$, given a configuration of Y's parents $\pi(Y) = c$, we may define a mapping such $p(Y = y|\pi(Y) = c) = 1$. DTs in GMTK natively support complicated expressions of parent variables involving max, min, abs, round, and basic arithmetic operators. Note that this provides a substantial amount of modeling power as well as affording efficient inference as $p(Y = y|\pi(Y) = c) = 1$ so that we need not waste compute considering other values $\pi(Y) \neq c$. For further information on DTs, please consult the official GMTK documentation.

Note that if you desired a conditional distribution which did change based on the value of $O(0)$'s parent variables, i.e., $p(O_t|X_t = x) = p(O_t) = \mathcal{N}(\mu, \sigma^2)$, it would suffice to exclude a mapping. In such a case, the distribution over $O(0)$ would be a single Gaussian for all time.

In GMTK, each Gaussian mean, variance, and mixture coefficient is defined in a separate data structure. The Gaussian components are then comprised of these constituent units. Finally, Gaussian collections are defined with elements of defined Gaussian components. We now describe each of these in detail.

2.0.1 Gaussian means, variances, and mixture coefficients

The means, variances, and mixture coefficients all follow a similar type declaration syntax we've previously seen for CPTs:

```
number of data instances to follow
data instance index
data instance name
data instance dimension
data instance values
```

The means, variances, and mixture coefficients are defined as follows in `hmm_gaussLeaves.mtr`:

```
MEAN_IN_FILE inline
3
0
mean0 1 0.5
1
mean1 1 0.5
2
mean2 1 0.5

COVAR_IN_FILE inline
3
0
covar0 1 1.0
1
covar1 1 1.0
2
covar2 1 1.0

DPMF_IN_FILE inline
1
0
unityDPMF
1
1.0
```

The text `MEAN_IN_FILE inline`, `COVAR_IN_FILE inline`, and `DPMF_IN_FILE inline` are reserved words notifying the GMTK parser that means, covariances, and DPMFs, respectively, are defined following the command in the current file being read. These quantities may also be declared separately in an external file. For instance, `MEAN_IN_FILE means.txt ascii` would specify to read means from a file `means.txt` written in ascii. The file may then hold the definition of the vector of 3 means defined above. The mixture coefficients are defined in terms of a *Dense probability mass function* (DPMF) where we may reuse the same mixture coefficient, whose value is unity, since we are only considering scalar Gaussians. When learning mixtures of Gaussians with $n > 1$, one must be cautious of reusing mixture coefficients, as they may actually be quantities which should be individually learned (i.e., declared differently). This will, of course, vary depending on the application.

2.0.2 Gaussian components

Now that we've defined the basic Gaussian building blocks of means, variances, and mixture coefficients, we may define actual Gaussians in the form of Gaussian components (scalar Gaussians herein). The syntax is:

```
number of components to follow
component index
component dimensionality
component type
component name
mean vector name
covariance vector name
```

Gaussian component syntax is fairly similar to what we've seen before. We begin by defining the number of components to be defined, followed by the current component index. Now, different from before, we first define the component dimensionality, then the component type, then the component name. The component type may be one of the following: (0) defined by means and covariance; (1) defined by means, covariance, and dlink structure (not covered). Finally, we the mean and covariance names for this components. Note that the dimensionality of the mean vector and covariace vector/matrix (vector for diagonal Gaussians) must be the same as the component dimensionality. From `hmm_gaussLeaves.mtr`, we have:

```
MC_IN_FILE inline
3
0
1
0
gc0 mean0 covar0
1
1
0
gc1 mean1 covar1
2
1
0
gc2 mean2 covar2
```

where the text `MC_IN_FILE inline` tells the GMTK parser that Gaussian mixture components are to be subsequently defined.

2.0.3 Gaussian mixtures

Now we may define mixtures of Gaussian components. The syntax is:

```
number of componens to follow
mixture index
mixture dimensionality
mixture name
number of components in mixture
DPMF name
component names
```

From `hmm_gaussLeaves.mtr`, we have:

```
MX_IN_FILE inline
3
0 1 mixture0 1 unityDPMF gc0
1 1 mixture1 1 unityDPMF gc1
2 1 mixture2 1 unityDPMF gc2
```

where, as we've similarly seen before, the text `MX_IN_FILE inline` tells the GMTK parser that Gaussian mixtures are to be subsequently defined.

2.0.4 Gaussian collections

Finally, we are ready to build our collection of mixtures to model $p(O(0)|X(0) = i) = v(i)$. The syntax for a collection is:

```
number of collections to follow
collection index
collection name
collection dimensionality
mixture name per state
```

From `hmm_gaussLeaves.mtr`, we have:

```
NAME_COLLECTION_IN_FILE inline
1
0
pO_given_X
3
mixture0
mixture1
mixture2
```

where, as we've similarly seen before, the text `NAME_COLLECTION_IN_FILE inline` tells the GMTK parser that Gaussian collections are to be subsequently defined.

2.1 Training Gaussian parameters

Now we are finally ready to train the model! The training script is available as `train_gaussLeaves.sh`. The options do not change from Section 1.1, save for defining the output to be `-inputMasterFile` `hmm_gaussLeaves.mtr`.

2.1.1 Testing Gaussian parameters

The test script is similar as before, available as `test_gaussLeaves.sh`. Now, testing the above EM trained Gaussian and multinomial parameters over 15000 test samples, we obtain:

```
Correctly identified 5000 of 15000 data points
```

This certainly seems much worse than the performance achieved when our observations were boolean. Indeed, we've achieved accuracy no better than random. It turns out that with our observations now being real valued, the model complexity has increased significantly. To effectively train in the face of this increased complexity, we utilize labeled training data (described in Section 2.2) wherein, for each observed sample, we know its true label. In this case, such training is said to be *supervised*.

2.2 Supervised training

For the training data, assume that we not only have the observations, but we also have the correct observation labels. In the case of the weather problem with real valued observations, this amounts to a collection of days wherein your coworker not only gives you a noisy estimate of that days weather, but also whether that day was sunny, rainy, or foggy. For such days, our hidden nodes are actually observed, as seen in Figure 3. Such a model specifically designed for training is often called a *boot* model, as this model is used to train the parameters of the model to be used for test time. When doing Viterbi decoding, the test model is typically referred to as a *decoder*.

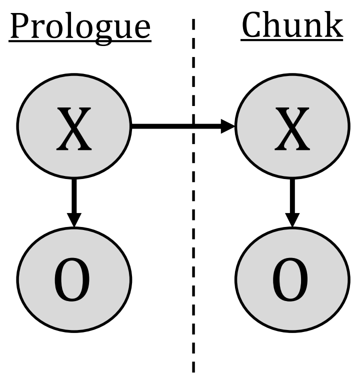


Figure 3: HMM for supervised training.

As seen in `hmm_gaussLeaves_train.str`, our structure changes to

```
frame: 0 {
variable : X {
```

```
type : discrete observed 1:1 cardinality 3;
conditionalparents : nil using DenseCPT("pX");
  }

variable : O {
type : continuous observed 0:0;
conditionalparents : X(0) using mixture collection("pO_given_X") mapping("internal:copyParent");
  }

}

frame: 1 {
variable : X {
type : discrete observed 1:1 cardinality 3;
conditionalparents : X(-1) using DenseCPT("pX_given_X");
  }

variable : O {
type : continuous observed 0:0;
conditionalparents : X(0) using mixture collection("pO_given_X") mapping("internal:copyParent");
  }

}
```

so that $X(0)$ is now observed. Note that the observation for $X(0)$ is the second entry in the global observation matrix. This is important since, in GMTK, real valued observations must come before integer valued observations. The script `train_gaussLeaves.sh` has the pertinent commands. To train using the boot model, change `BOOT=1` to `BOOT=0` (line 4). The relevant changes between `train_gaussLeaves.sh` and our previous training script are listed below.

```
    -of1 $TRAIN -fmt1 ascii -nf1 1 -ni1 0 \
    -of2 $LABELS -fmt2 ascii -nf2 0 -ni2 1 \
```

These changes load the real valued observation file, variable `TRAIN`, first then the integer valued observation labels, variable `LABELS`. Thus, real valued observations occur first in the global observation matrix, followed by the integer valued observations. Note that the current files include 1500 labelled training instances. Testing with the resulting trained parameters, `trained_gaussLeaves.params`, using `test_gaussLeaves.sh` results in

```
Correctly identified 10374 of 15000 data points
```

Thus, we achieve 69.16% accuracy, a substantial improvement from our earlier, unsupervised training.

2.2.1 Discriminative training via maximum mutual information estimation

GMTK supports discriminative training via *maximum mutual information* (MMI) estimation [2], also called conditional maximum likelihood estimation. Such training has the potential to learn improved parameters relative to generative training (such as EM). MMI estimation not only maximizes the parameters with respect to the supervised log-likelihood (as is done in EM) but simultaneously learns parameters which minimize the log-likehood of hypotheses not equal to the supervised labels (i.e., a background set of "incorrect" labels). In practice, two models are necessary; the *numerator* model, corresponding to the boot model, and the *denominator* model, corresponding to the decoder. The relevant script is `discTrain_gaussLeaves.sh`. Learning parameters using this script and testing, we obtain

```
Correctly identified 10457 of 15000 data points
```

Thus, discriminative training results in 69.71% accuracy, a nice improvement.

In GMTK, the MMI objective is optimized using *stochastic gradient ascent* (SGA) [3]. Critical to effective discriminative training are the SGA learning rate parameters, listed below from `discTrain_gaussLeaves.sh`.

```
    -updateMean T \
    -updateCovar T \
    -updateDPMF T \
    -updateCPT F \
    -useAdagrad F \
    -covarLr 1.0e-6 \
    -meanLr 1.0e-6 \
    -initLr 1.0e-6 \
    -useCovarDecayLr T \
    -useMeanDecayLr T \
    -useDecayLr T \
    -decayCovarLrRate 1.0 \
    -decayMeanLrRate 1.0 \
    -decayLrRate 1.0 \
```

The `-update` switches specify whether means, covariances, DPMFs, or CPTs are learned. The `-covarLr, -meanLr, -initLr` switches specify the learning rate for the Gaussian covariances, Gaussian means, and all DPMFs/CPTs, respectively. The `-useCovarDecayLr, -useMeanDecayLr, -useDecayLr` switches specify whether the learning rate decays in subsequent iterations as the optimization proceeds. Finally, the `-decayCovarLrRate, -decayMeanLrRate, -decayLrRate` switches specify the rate of decay where, defining a parameters initial learning rate as l and the decay rate as r, the learning rate in iteration i will thus be l/i^r.

2.2.2 Discriminative training with AdaGrad

Determining a good learning rate schedule for SGA is not a simple task. GMTK thus supports the adaptive gradient algorithm, AdaGrad [4], wherein the learning rate is automatically adjusted in subsequent iterations. When using AdaGrad, the only learning rate parameters which matter are the initial learning rates. Enabling AdaGrad and setting the initial learning rate to the following:

```
    -useAdagrad T \
    -covarLr 1.0e-2 \
    -meanLr 1.0e-2 \
    -initLr 1.0e-2 \
```

we achieve the following performance after 3 iterations:

```
Correctly identified 10525 of 15000 data points
```

We now achieve 70.17% accuracy on the classification task of interest, further improving upon the discriminative training results with decaying learning rates.

2.2.3 Discriminative training via deep neural networks

GMTK also supports supervised training of deep neural networks (DNNs)[5]. For our HMM, this means the emission probabilities will change to that of a neural network whose weights and biases are learned via backpropagation. The observations in the structure file change as follows (seen in `hmm_dnn.str`)

```
variable : O {
type : discrete observed value 1 cardinality 2;
conditionalparents : X(0) using DeepVirtualEvidenceCPT("deepVe");
  }
```

The DNN using virtual evidence, wherein $O(0)$ is called a *virtual evidence child*. A virtual evidence child, in GMTK, has a single parent, has cardinality 2, and is observed to value 1. The virtual evidence function governing the child, which is $f(O(0), X(0)) = p(O(0) = 1|X(0) = x)$, may be any non-negative function (even unnormalized, offering a large degree of modeling flexibility). The virtual evidence function may thus be thought of as a weighting function for all hypotheses of the virtual evidence parent (there must be only a single virtual evidence parent in GMTK) such that the relative weight of a hypothesis $X(0) = x$ plays the most critical factor during inference; that is, for $X(0) \in \{x_0, x_1\}$, $f(O(0), x_0) > f(O(0), x_1)$ favors hypotheses where $X(0) = x_0$. The DNN itself is declared by the `DeepVirtualEvidenceCPT` "deepVe". A DeepVirtualEvidenceCPT in GMTK is optimized for the possibly many matrix multiplies required of a feed forward pass during decoding.

The most significant changes in defining the DNN occurs in the master file, `hmm_dnn.mtr`. First, constants are included in the file `hmm_dnn.h` via

```
#include "hmm_dnn.h"
```

Next, the matrices per layer, are defined:

```
DOUBLE_MAT_IN_FILE dnn ascii
```

The file `dnn` contains the matrices in ascii format. The matrix definition format is

```
number of matrices to follow
matrix index
matrix name
number of matrix rows
number of matrix columns
matrix values in raster order
```

Next, the DNN is defined infile as

```
DEEP_NN_IN_FILE inline 1
```

```
0
dnn                  % Deep NN name
NUMFEATURES
3                    % cardinality of labels to be predicted
matrices:4 10 10 10 3
matrix0:g0           % name of double matrix (weights) for layer 0 (input layer)
squash0:rectlin      % non-linearity for layer 0
matrix1:g1
squash1:rectlin      % options are softmax, logistic, tanh, oddroot, linear, rectlin
matrix2:g2
squash2:rectlin
matrix3:g3           % output layer
squash3:softmax      % ensure outputs sum to 1
END
```

Here we've defined a four layer network where the first layer, g0, is the input layer (the real valued temperature observations, for our task at hand) consisting of 10 rectified linear neurons (10 observed units). The remaining layers are the hidden units consisting of 10 rectified linear, 10 rectified linear, and 3 softmax units. Note that, in the definition for the matrices in file dnn, each layer has an extra column which corresponds to that layer's bias. For instance, g0 has 10 rows corresponding to the 10 input units and 2 columns, one for each neuron's weight and the other for the bias.

Finally, the deep virtual evidence function defined as

```
DEEP_VE_CPT_IN_FILE inline 1
0                    % CPT #
deepVe               % CPT name
1                    % # of parents (must be 1 for virtual evidence)
3                    % parent cardinality (must = # NN outputs)
2                    % self cardinality (must be 2 for virtual evidence)
dnn                  % name of deep NN that computes probabilities of parent
f_offset:0           % starting index in observation vector for input features
nfs:1                % # of input features to take from each frame
radius:RADIUS        % use 2 * radius + 1 = 9 frames as NN input;  t-r : t+r
END
```

The comments do well to describe the parameters. The radius parameter defines the number of inputs, from the observation matrix, to concatenate together in the input layer. This is often important for temporal signals where correlation exists between adjacent observations, such as in a speech waveform. For our particular example, the sequence of inputs does not constitute a wafeform, so the radius is defined to be 0 in hmm_dnn.h.

We are now ready to train the network. The training script is train_dnn.sh, where the pertinent GMTK call being

```
RADIUS=0
NUMFEATURES=$((2*RADIUS+1))

gmtkDMLPtrain \
  -of1 $TRAIN -fmt1 ascii -nf1 1 -ni1 0 \
  -of2 $LABELS -fmt2 ascii -nf2 0 -ni2 1 \
  -startSkip $RADIUS -endSkip $RADIUS -constantSpace T \
  -trainingSched permute \
  -inputMasterFile hmm_dnn.mtr \
  -deepMLPName dnn -labelOffset 1 \
  -featureOffset 0 -numFeatures $NUMFEATURES -radius $RADIUS \
  -pretrainType AE -ptNumEpochs 0.2 -ptNumAnneal 0.1 -ptL2 1e-3 \
  -bpNumEpochs 4415 -bpNumAnneal 0.15 -bpL2 1e-3 \
  -outputMasterFile dnn.params \
  -batchQueueSize 524300 -pretrainType none \
  -bpInitStepSize 2.000000e-03 \
  -ptMiniBatchSize 100 -bpMiniBatchSize 100 -allocateDenseCpts 2
```

The DNN is trained using stochastic gradient descent (SGD), the interpretation of which is almost completely analogous to SGA (used in MMI). The parameter `-bpNumEpochs` defines the number of training epochs and `-bpMiniBatchSize` the number of training instances to consider before taking an SGD step within an epoch. Thus, setting `-bpMiniBatchSize` to 100 with 1500 training means we consider 100 observations when computing a gradient and take 15 steps per each epoch. For this example, we've turned off pretraining [6], though such parameters (beginning with `pt`) have equivalent interpretations to their backpropagation counterparts (which begin with `bp`). The initial learning rate is defined by `-bpInitStepSize` and follows a fixed annealing schedule following the declaration of `-bpNumAnneal`. Training using `train_dnn.sh` and testing using `test_dnn.sh`, we achieve the following performance:

```
Correctly identified 10535 of 15000 data points
```

so that we now achieve 70.23% accuracy on the classification task.

The accuracy for the various forms of training discussed herein are summarized in Table 1.

Table 1: Different training methods given 1500 labeled training instances, tested over 15000 instances.

Training procedure	# correctly classified	% correctly classified
EM (unlabeled)	5000	33.33
EM	10374	69.16
MMI (decaying lr, 3 iterations)	10457	69.71
MMI (adagrad, 3 iterations)	10525	70.17
DNN	10535	70.23

References

[1] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the royal statistical society. Series B (methodological)*, pp. 1–38, 1977.

[2] D. Povey and P. C. Woodland, "Improved discriminative training techniques for large vocabulary continuous speech recognition," in *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*, vol. 1, pp. 45–48, IEEE, 2001.

[3] L. Bottou, “Stochastic gradient descent tricks,” in *Neural Networks: Tricks of the Trade*, pp. 421–436, Springer, 2012.

[4] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” *The Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.

[5] Y. Bengio, I. J. Goodfellow, and A. Courville, “Deep learning.” Book in preparation for MIT Press, 2015.

[6] G. E. Hinton and R. R. Salakhutdinov, “Reducing the dimensionality of data with neural networks,” *Science*, vol. 313, no. 5786, pp. 504–507, 2006.