Speech Recognition for the MessagePad 2000

by Stephen Breit and Bent Schmidt-Nielsen,
Dragon Systems, Inc. 320 Nevada St., Newton, MA 02160

Introduction

The MessagePad 2000 has two key ingredients which make Newton, for the first time, a viable platform for portable speech recognition. One is the powerful 160 MHz StrongARM microprocessor, and the other is high quality, 16-bit audio input. With the MessagePad 2000 as an enabler, the potential benefits of providing speech recognition on Newton devices are clear. First, in mobile situations, a speech user interface (SUI) can free one or both hands for other uses, such as handling material or operating equipment. Second, navigating a complex application can be much faster with speech than with a pen because speech input need not be constrained by the tree-like structure of a GUI. And third, as an input modality, speech is much faster than handwriting. The use of speech input on small devices such as the Newton **platform** is particularly compelling because they **may** either lack keyboards, or have keyboards that are too small to use efficiently. Or putting it more succinctly "Computers keep getting smaller, but our fingers stay the same size".

By now you are conjuring up visions of holding a Newton **device** and having it recognize and understand whatever you say. Providing such a capability is, of course, our ultimate aim, but we can safely say that this is at least a few years away. In the mean time, we can offer constrained speech recognition capabilities which will be a valuable addition to many applications. Users will be able to say phrases and sentences in a natural way, i.e. without pausing between words, but the vocabulary and word order will be constrained to a pre-defined grammar. For example, for an inventory application, the user will be able to say "Part number one five three seven", or, more generally, "<qualifier><digit_string>", where <qualifier> might be "Quantity", "Part number", "UPC", etc. and <digit_string> is a series of 1 to 12 digits. Examples of other applications which might benefit from this type of speech recognition capability include medical record keeping, insurance appraisal, meter reading, rental car returns, sports data acquisition, and law enforcement. And there will be many others.

In this article, we describe our efforts to port one of Dragon's speech recognition engines to **Newton**, and the capabilities which we expect to offer to Newton software developers. Since this is the first time that

anyone has done speech recognition on the Newton **platform**, and speech recognition will undoubtedly be new to many readers, we begin with an overview of a typical speech recognition system. This provides a basis for understanding the capabilities of the speech recognition system that we have ported to **Newton**. Next, we describe some of the challenges to doing the port, and how we have overcome them. Then, we describe a grammar for an inventory application. And finally, we provide some code snippets which illustrate how you include speech recognition in an application.

Overview of a Speech Recognition System

The majority of commercially available speech recognition systems rely on Hidden Markov Models (HMMs) and have the key components shown in Figure 1. To explain how this system works, we begin at the lower left with the microphone. The analog signal from a microphone is converted to a digitized wave form by the audio system hardware. This *audio input* is processed by the software *audio analysis* module which converts it to a form suitable for speech recognition. To do this, the audio analysis module applies a series of transforms, starting with an FFT, and outputs a *frame* of parameters every 20 ms. Typically, each frame contains 12 to 36 *parameters.* The audio analysis module also may apply a the speech detection algorithm to detect transitions from silence to speech, and from speech to silence. A collection of consecutive frames which begins and ends with silence is known as an *utterance.*

Figure 1: The components of a typical speech recognition system.

Before describing what the *recognizer* does with the utterance, we must describe the other key inputs. The *acoustic models* are built by collecting a set of utterances of similar sounds and assembling them into

what amounts to a prototypical utterance for each sound. The individual sounds can be either elemental sounds called *phones* (hence *phonetic* modeling) or entire words (hence *whole-word* modeling). The accuracy of a speech recognition system is largely dependent on the quality of the acoustic models. Models that are intended for use by speakers who have not trained the recognizer are said to be *speaker independent.* Models that are intended for a specific speaker, and have generally been trained by that speaker, are said to be *speaker dependent.* Both types of models can be further *adapted* to a particular speaker, and this generally improves recognition accuracy.

If phonetic models are used, the *dictionary* provides a translation between word spellings and their pronunciations in terms of phones. If whole-word models are used, the dictionary simply provides a mapping between a word and its acoustic model. The dictionary and acoustic models are usually supplied with the speech recognition system, but it is up to the application developer to supply the *grammar.* The grammar defines the sequences of words that can be recognized. Some examples of grammars are given later in this article.

Simply put, the *recognizer* processes each utterance and returns the sequence of words that was most likely to have produced the utterance. Going into a bit more detail, the application specifies an active grammar, either from a predefined data structure, or by building it "on the fly". Then the application tells the audio analysis module to begin processing audio input. When the audio analysis module detects the start of an utterance, the recognizer begins its job. To start with, the recognizer hypothesizes all words that could have started the utterance based on the active grammar. It then *scores* each frame in the utterance against these hypotheses, and the score decreases with each successive frame. The *score* is the log probability that the acoustic model could have produced the observed utterance; thus, the hypothesis with the highest score had the highest probability of producing the observed audio input. As the recognizer scores successive frames against all active hypotheses, it prunes hypotheses whose scores are much lower than the best-scoring hypothesis in order to save computations.

The Speech API provides a well defined interface by which the application interacts with the recognition engine. The complexity of the Speech API depends on the recognition capabilities being offered. It may have as few as 20 entry points for simple command and control capabilities, or more than 200 calls to support dictation.

Speech Recognition Capabilities for Newton

With some background information in hand, we can now describe the capabilities of the speech recognition engine that we have ported to **the**

Newton. Dragon has a number of recognition engines in its stable; we chose one that is known internally at Dragon as C-REC. C-REC was originally developed to run on a digital signal processor (DSP), so it has a small memory footprint, and the code is relatively small and portable.

C-REC can recognize phrases and sentences that are spoken continuously, *i.e.* in a natural way without pausing between words. Though C-REC imposes no restrictions on the size of the vocabulary or the grammar, it is best suited to "small-vocabulary" recognition tasks with simple context-free grammars. The vocabulary and grammar can be context sensitive. "Small vocabulary" in this case means that the grammar *perplexity*, or average branching factor, should be approximately 50 or lower in order to achieve real-time performance. The actual vocabulary size may be quite large (thousands of words). Higher perplexity grammars can be used if a response time of a second or more is acceptable.

C-REC works with either whole-word or phonetic models. To build speaker-independent whole-word models, we need one sample of each word from many (100 or more) different male and female speakers. This is a significant disadvantage relative to phonetic models where, once the models are built, any word for which we have a phonetic pronunciation can be recognized. On the other hand, experiments with C-REC on limited vocabularies have shown that whole-word models are more accurate than phonetic models. And whole-word models require less computation, at least when the active vocabulary is relatively small. Due to these considerations, plus some additional factors which are discussed in the next section, we chose speaker-independent, whole-word models for the initial port of C-REC to **the** Newton. We envision offering a software developer's kit which includes whole-word, speaker-independent models for a "standard" vocabulary. With a judicious choice of words, this standard vocabulary should be sufficient for a number of applications. If a particular application requires words that are not in the standard vocabulary, we will record samples of the additional words and build a set of custom acoustic models for that application.

We are inevitably asked about recognition accuracy. The answer to this question depends on many factors, including the size of the active vocabulary, the grammar, the quality of the microphone, the amount of background noise, and the quality of the acoustic models. Just to give some idea, we have estimated from test results that, in quiet conditions at least, continuously spoken strings of 5 digits should be recognized completely correctly 98% of the time.

Implementation on the MessagePad 2000

Next, we describe some details of how we ported C-REC to the MessagePad 2000. Our first step was to measure the quality of the audio system, since we have found a wide variation in the quality of the audio hardware on notebook computers. As we expected, the signal from the built-in microphone is not of sufficient quality to do accurate speech recognition. But we were pleased to discover that we got a high quality audio signal when we connected a head-mounted, noise-canceling microphone to the MessagePad via a small, custom-built amplifier and the line-in pins on the Newton connector.

C-REC is written in C, so we have been using a beta version of the Newton C/C++ Toolkit to port it to the Newton **platform**. The Newton C/C++ Toolkit does not allow any global or static variables. Therefore, we had to remove all such variables from C-REC, and store pointers to persistent data in a *binary object*. This proved to be a laborious process because the Newton C/C++ compiler only indicates that there are global symbols, but does not give their names.

Before porting the audio analysis module, we had to convert from floating-point to fixed-point operations because the StrongARM does not have floating-point instructions. We verified that there was no loss in recognition accuracy from this change. When we ran the fixed-point audio analysis on the Newton **device**, we found that it required only 6% of the CPU cycles. This was very encouraging because the audio analysis module must run continuously whenever the microphone is "turned on", whereas the recognizer runs only when there is speech input to be processed.

In order to port the recognizer itself, we had to find a way to store and load the acoustic models. One approach is to store the models in a binary object, and pass this binary object to the recognizer. Due to the NewtonScript garbage collection, the binary object would periodically move around in the memory system. This would require patching up pointers to data in the binary object each time the recognizer is called. An alternative approach, which was expedient and gave better performance, was to compile the acoustic models into the code. The disadvantages of this approach are that it takes a lot of memory to do the compilation, and that the code must be recompiled each time we change the acoustic models. As of this writing, we are compiling the models into the code. The memory required for the code alone is approximately 100 kB. For most applications, the additional memory required for the acoustic models will be between 250 kB and 1MB. During recognition, the recognizer requires an additional 250 kB of dynamically allocated memory.

A Speech Interface for an Inventory Application

Suppose you want to develop a speech-enabled application for taking inventory in a store or warehouse. Let's assume that four pieces of information are needed to inventory an item: 1) its location in the warehouse, expressed as a row number and a section number, 2) its 6-digit number, 3) its color, and 4) the quantity of identical items. You design a view which has fields for entering this information. You would like the user to be able to open a new inventory view by voice and enter data in the fields in any order.

To illustrate the design of the speech interface, we need to introduce a pseudo code for expressing a grammar. The grammar has three basic entities: *words, rules,* and *groups.* A word may be either a single word or a phrase consisting of two or more words. We denote a word simply be spelling out the word. We denote a phrase by putting underscores between the component words. If we want the recognizer to return text that is different than the spelling of the word, we enclose the return text in parentheses following the word. For example, "two(2)" indicates that the recognizer should return the character "2" when it hears a word that sounds like "two".

A group is a collection of words or rules. We denote a group by a comma-separated list of words or rules enclosed in curly brackets. We denote the name of a group by text enclosed in angle brackets. For example,

<global> = {next_item, previous_item, go_back, start_over, enter_data};

defines a group named "global" in which any of the phrases listed on the right-hand side can be recognized. Each phrase counts as one word. We purposely called this rule "global" because these words will always be active in the inventory application. For example, the user will always be able to return to the previous field on the view by saying "go back".

To enter numbers between 1 and 99, it is useful define the following groups (we use ellipses to fill out an obvious sequence of words):

<one2nine> = { one(1), two(2), three(3),..., nine(9)};
<digit> = { zero(0), oh(0), <one2nine>};
<ten2nineteen> = {ten(10), eleven(11), twelve(12),...,nineteen(19)};
<twenty2ninety> = {twenty(20), thirty(30), forty(40),...,ninety(90)};

Now we need to introduce rules. A rule is a sequence of words and/or states. We denote the name of a rule by text enclosed in square brackets. For example, the rule

[twentyone2ninetynine] = <twenty2ninety><one2nine>;

allows users to say numbers between “twenty-one” and “ninety-nine”, excluding multiples of 10. Finally, the group

<number> = {<one2nine>, <ten2nineteen>, <twenty2ninety>, <twentyone2ninetynine>};

allows users to say, in a natural way, any number between one and 99. Note that the <number> group is defined in terms of previously defined groups.

For the inventory application, it will also be useful to define the group

<color> = {black, white, red, blue, green, yellow, orange, purple}

and the following rules:

[location] = row <number> section <number>;
[part_number] = part_number <digit><digit><digit><digit><digit><digit>;
[color] = color <color_name>;
[quantity] = quantity <number>;

Finally, we define a group which encompasses all of the rules for the inventory application:

```
<inventory_view> = {[location], [part_number],
            [color], [quantity], <global>};
```

When this rule is active, the user may enter data in any field on the form at any time, return to the preceding field if there is an error in it, clear the entire form, return to the previous item, or move on to the next item.

This example gives some idea of how you define a grammar. There are a total of 47 words in this example (each phrase counts as one word). We expect that all of these words will be part of our standard vocabulary.

Programming Example

This example shows how you incorporate speech recognition in an application:

Initializing the speech recognition system

The first step is to instantiate the speech recognition system.

```
SpeechRecog := {
  // put additional variables or overrides here
  _proto: protoDragonSpeechRecognizer;
}
local theView := self;
SpeechRecog:setUp( theView, inputSource, errorCallBack );
```

The `inputSource` argument determines whether the system takes audio input from the built-in microphone, or line-input. You must provide a function for error callbacks. It is desirable to execute this code when

your application starts up, otherwise the user may experience a brief delay while the systems allocates memory for the audio input buffers.

Defining the grammar

At this point, you need to define the grammar from which you want to recognize. We have not implemented the calls for defining a grammar in NewtonScript yet, so we cannot offer any code fragments here. Suffice it to say that there will be a set of calls for defining groups and rules as they are described in the previous section. And there will be calls for iterating the words in the dictionary, and checking whether a particular word is in the dictionary.

Starting the audio analysis module

The next step is to start the audio analysis module:

```
SpeechRecog:startAudio();
```

This could have been done transparently by `setUp()`, but there is a good reason for giving you control over when it starts. Once started, the audio analysis module must run continuously. This draws power and, more importantly, overrides the power-down features of the MessagePad 2000. You need to start the audio analysis module well before you expect speech input, because it takes almost a second for the MessagePad 2000 to charge up a capacitor in the audio system hardware.

Starting and stopping the recognizer

You define one or more callback functions for processing the results from the speech recognizer. For example, you may have a different callback function for each view or context. The `transcription` parameter is a text transcription of what was said (it must be permitted by the grammar, of course). The `grammarInfo` parameter is a reference to a NewtonScript frame which allows you to determine which rule was recognized.

```
resultCallBack := func( transcription, grammarInfo )
  begin;
  setValue(resultView, 'text, transcription);
  end;
```

You start the recognizer:

```
SpeechRecog:startRecog( resultView, resultCallBack, grammarGroup );
```

Once started, the recognizer starts recognizing whenever a new utterance is available and does a callback when it is finished recognizing the utterance. You must stop the recognizer when you want to change the grammar or as the first step in the process of shutting down the speech recognition system:

```
SpeechRecog:stopRecog();
```

Stopping the audio analysis module

After shutting down the recognizer, you shut down the audio analysis module:

```
SpeechRecog:stopAudio();
```

Shutting down the speech recognition system

Finally, unlike an ordinary NewtonScript program, you must free the resources used by the recognition system before assigning nil to the Recog variable:

```
SpeechRecog:shutDown();
SpeechRecog := nil;
```

Conclusions

As a result of work by Bent Schmidt-Nielsen, Maha Kadirkamanathan, and Shaun Keller, a small-vocabulary, continuous-speech recognizer is now running on the Newton **platform**. As of this writing (early February '97), it recognizes strings of continuously spoken digits and returns a text transcription within one-half second after the user has finished speaking. It is very exciting to get this responsiveness from a computer that is powered by a few AA cells! In the immediate future, we plan to supplement the digits vocabulary with a set of standard words and phrases such as those listed in the inventory application example. Further down the road, we will consider adding other features such as phonetic models, and the ability to adapt the speaker-independent models to an individual speaker.

Judging from the enthusiastic response we have already received from Newton developers who have heard about our efforts, there is a lot of pent-up demand for speech recognition on the Newton **platform**. We are planning to offer a tool kit which will enable Newton developers to speech enable their applications. The tool kit will include an AutoPart, a user "proto", documentation of the NewtonScript API, a microphone, and a preamplifier. Built into the AutoPart will be acoustic models for a standard vocabulary. If the standard vocabulary does not meet the needs of a particular application, we are prepared to develop custom acoustic models for that application. Beyond the inventory application that is suggested in this article, we believe that the type of capability we have described will be useful for many other applications. We look forward to hearing your requirements for speech recognition on the Newton **platform**.