1 Introduction

Finite-state automata have a relatively long history of application in machine learning. The majority of these applications involve sequential data. For example, they are or have been used in speech recognition, machine translation, protein function analysis, and other tasks in natural language and computational biology.

However, the application of these data structures in machine learning is far from main stream. In fact, their use decreased with the advent of end-to-end deep learning. However, the recent development of frameworks for automatic differentiation with automata suggests there may be renewed interest in the application of automata to machine learning.

This tutorial introduces weighted automata and their operations. Once this fundamental data structure is well understood, we then continue to build our intuition by working through some extended examples. At minimum, I hope that this tutorial engenders an appreciation for the potential of automata in machine learning. Ideally for some readers this tutorial will be a launching point for the incorporation of automata in machine-learning research and applications.

However, before launching into the more technical content, let's start with some broader perspectives in order to motivate the use of automata in machine learning.

1.1 Monolithic or Modular

In the past, complex machine-learning systems, like those used in speech recognition, involved many specialized hand-engineered components. The trend is now towards the opposite end of the spectrum. Most machine-learning applications involve a single, monolithic neural network. Both of these extremes have advantages, and both have disadvantages.

A primary advantage of automata in machine learning is their ability to harness the best of both worlds. Automata are capable of retaining many if not all of the advantages of a multi-component, hand-engineered system as well as those of a monolithic deep neural network. The next few paragraphs explain some of these advantages and the regime to which they apply.

Modular: One of the advantages of multi-component, hand-engineered systems over monolithic neural networks is modularity. In traditional software design modularity is a good thing. Modular systems are easier to develop since part of the system can be changed without needing to change the rest. In machine-learning systems, modularity is useful to avoid retraining the entire system when only

part of the model needs to be updated. Modularity can also be useful when the individual modules can be reused. For example, speech recognition systems are built from acoustic models and language models. Acoustic models can be language agnostic and used for different languages. Language models are general text based models which can be used in many different tasks other than speech recognition.

Compound errors: A primary disadvantage of modular systems is that errors compound. Each module is typically developed in isolation and hence unaware of the types of errors made by the modules from which it receives input. Monolithic systems on the other hand can be thought of as being constructed from many sub-components all of which are jointly optimized towards a single objective. These sub-components can learn to compensate for the mistakes made by the others and in general work together more cohesively.

Adaptable: Modular systems are typically more adaptable than monolithic systems. A machine-learning model which is tuned for one domain usually won't work in another domain without retraining at least part of the model on data from the new domain. Monolithic neural networks typically require a lot of data and hence are difficult to adapt to new domains. Modular systems also need to be adapted. However, in some cases only a small subset of the modules need be updated. Adapting only a few sub-modules requires less data and makes the adaptation problem simpler.

Learn from data: One of the hallmarks of deep neural networks is their ability to continue to learn and improve with larger data sets. Because of the many assumptions hard-wired into more traditional modular systems, they hit a performance ceiling much earlier as data set sizes increase. Retaining the ability to learn when data is plentiful is a critical feature of any machine-learning system.

Prior knowledge: On the other hand, one of the downsides of deep neural networks is their need for large data sets to yield even decent performance. Encoding prior knowledge into a model improves sample efficiency and hence reduces the need for data. Encoding prior knowledge into a deep neural networks is not easy. In some cases, indirectly encoding prior knowledge into a neural network can be done, such as the translation invariance implied by convolution and pooling. However, in general, this is not so straightforward. Modular systems by their very nature incorporate prior knowledge for a given task. Each module is designed and built to solve a specific sub-task, usually with plenty of potential for customization towards that task.

Modular and monolithic systems have complementary advantages with respect to these four traits. Ideally we could construct machine-learning models which retain the best of each. Automata-based models will take us a step closer towards this goal. However, to use automata to their full potential we have to overcome a couple of challenges. The key is enabling the use of weighted automata in training the model itself. This requires 1) efficient implementations and 2) easy to use frameworks which support automatic differentiation.

1.2 Advantages of Differentiable Automata

A key to unlocking the potential of automata in machine learning is enabling their use during the training stage of machine-learning models. All of the operations I introduce later are differentiable with respect to the arc weights of their input graphs. This means that weighted automata and the operations on them can be used in the same way that tensors and their corresponding operations are used in deep learning. Operations can be composed to form complex computation graphs. Some of the weighted automata which are input to the computation graph can have parameters which are learned. These parameters can be optimized towards an objective with gradient descent.

Automatic differentiation makes computing gradients for complex computation graphs much simpler. Hence, combining automatic differentiation with weighted automata is important to enabling their use in training machine-learning models.

Sophisticated machine-learning systems often separate the training and inference stages of the algorithm. Multiple models are trained in isolation via one code path. For prediction on new data, the individual models are combined and rely on a different code path. The constraints of the two regimes (training and inference) are such that separation from a modeling and software perspective is often the best option. However, this is not without drawbacks.

First, from a pragmatic standpoint, having separate logic and code paths for training and inference requires extra effort and usually results in bugs from subtle mismatches between the two paths. Second, from a modeling standpoint, optimizing individual models in isolation and then combining them is sub-optimal.

One of the benefits of combining automatic differentiation with weighted automata is the potential to bring the training and inference stages closer together. For example, speech recognition systems often uses hand-implemented loss functions at training time. However, the decoder (used for inference) brings together multiple models represented as automata (lexicon, language model, acoustic model, *etc.*) in a completely different code path. By enabling automatic differentiation with

graphs, the decoding stage can also be used for training. This has the potential to both simplify and improve the performance of the system.

Combining automatic differentiation with automata creates a separation of code from data. Loss functions like Connectionist Temporal Classification, the Automatic Segmentation criterion, and Lattice-Free Maximum Mutual Information have custom and highly optimized software implementations. However, these loss functions can all be implemented using graphs and (differentiable) operations on graphs. This separation of code from data, where graphs represent the data and operations on graphs represent the code, has several benefits. First, the separation simplifies the software. Second, the separation facilitates research by making it easier to experiment with new ideas. Lastly, the separation enables the optimization of graph operations to be more broadly shared.

1.3 Comparison to Tensors

Modern deep learning is built upon the tensor data structure and the many operations which take as input one or more tensors. Some of the more common operations include matrix multiplication, two-dimensional convolution, reduction operations (sum, max, product, *etc*), and unary and binary operations.

Automata are an alternative data structure and the operations are quite different in general. However, one can draw a loose analogy between the categories of operations with automata and those with tensors. Table 1.1 shows some of the common operations on tensors and their analogous operations on automata. The analogy is quite loose, but still useful at the very least as a mnemonic device and perhaps can help build intuition for the various operations on graphs.

For example, superficially the formula for matrix multiplication and transducer composition are quite similar. Assume we have three matrices such that $\mathbf{C} = \mathbf{AB}$. The element at position (i, j) of $\mathbf{C}$ is given by:

$$C_{ij} = \sum_k A_{ik} B_{kj}. \tag{1}$$

Assume we have three transducers (graphs) where $\mathcal{C}$ is the composition of $\mathcal{A}$ and $\mathcal{B}$, then the score of the path pair $(\mathbf{u}, \mathbf{v})$ is given by:

$$\mathcal{C}(\mathbf{u}, \mathbf{v}) = \operatorname*{LSE}_{\mathbf{r}} \mathcal{A}(\mathbf{u}, \mathbf{r}) + \mathcal{B}(\mathbf{r}, \mathbf{v}), \tag{2}$$

where LSE is the *log-sum-exp* operation. Don't worry if the details are not clear – section 4 covers transducer composition. The point is that both operations,

transducer composition and matrix multiplication, accumulate over an inner variable the values from each of the inputs. In matrix multiplication this is the shared dimension of the matrices **A** and **B**. In graph composition the inner variable is the shared path **r**.

Table 1.1: The table shows loosely analogous operations between tensors and automata (acceptors and transducers).

Tensor	Automata
Matrix multiplication, convolution	Intersect, compose
Reduction ops (sum, max, prod, *etc.*)	Shortest distance (forward, Viterbi)
Unary ops (power, negation, *etc.*)	Unary ops (closure)
n-ary ops (addition, subtraction, *etc.*)	n-ary ops (concatenation, union)

A higher-level analogy to tensor-based deep learning can also be made. Modern machine-learning frameworks like PyTorch and TensorFlow (and their ancestors like Torch and Theano) were critical to the success of tensor-based deep learning. These frameworks include support for automatic differentiation. They also provide easy to use access to extremely efficient implementations of the core operations. In the same way, automata-based machine learning should benefit from frameworks with these features. We are just beginning to see new developments in frameworks for automata-based machine learning including GTN[1] and k2.[2] Perhaps these will encourage the use of automata in machine learning.

1.4 History and Bibliographic Notes

Hopcroft et al. [12] provides an excellent introduction to non-weighted automata. Mohri [15] gives a more formal and general treatment of weighted automata and associated algorithms.

Weighted finite-state automata are commonly used in speech recognition, natural language processing, optical character recognition, and other applications [6, 13, 14, 17, 18]. Pereira and Riley [20] developed an early application of weighted automata to speech recognition, though before that there were other applications in natural language processing [21, 23]. The Graph Transformer Networks of Bottou

[1] I am a co-author of the GTN framework which is open source at `https://github.com/gtn-org/gtn`

[2] The k2 framework is the successor of Kaldi and is open source at `https://github.com/k2-fsa/k2`

et al. [5], a similar though more general framework, were developed around the same time and applied to character recognition in images.

The sequence criteria mentioned in section 1.2, namely Connectionist Temporal Classification [9], the Automatic Segmentation criterion [8], and Lattice-free Maximum Mutual Information [22], are most commonly used in speech recognition. Section 6 shows how to implement a subset of these using weighted automata. Hannun [10] gives a more detailed introduction to Connectionist Temporal Classification.

In terms of software, two of the better known libraries for operations on WFSTs are OpenFST [16] and its predecessor FSM [3]. In section 1.3, I compared weighted automata to tensors. PyTorch [19] and TensorFlow [2] are two of the most used libraries for tensor-based deep learning with automatic differentiation. These were based on earlier frameworks including Torch [7] and Theano [4]. Libraries which support automatic differentiation with weighted automata have only recently been developed [1, 11].