An Introduction to Weighted Automata in Machine Learning

Awni Hannun*

October 14, 2021

Abstract

The goal of this work is to introduce the reader to weighted finite-state automata and their application to machine learning. I begin by motivating the use of automata in machine learning and proceed with an introduction to acceptors, transducers, and their associated properties. Many of the core operations of weighted automata are then described in detail. Following this, the work moves closer to the research frontier by explaining automatic differentiation and its use with weighted automata. The last section presents several extended examples to gain deeper familiarity with weighted automata, their operations, and their use in machine learning.

*Send correspondence to awni.hannun@gmail.com

Contents

1 Introduction **3**
1.1 Monolithic or Modular . . . 3
1.2 Advantages of Differentiable Automata . . . 5
1.3 Comparison to Tensors . . . 6
1.4 History and Bibliographic Notes . . . 7

2 Acceptors and Transducers **9**
2.1 Automata . . . 9
2.2 Acceptors . . . 10
2.3 Transducers . . . 14

3 Basic Operations **17**
3.1 Closure . . . 17
3.2 Union . . . 19
3.3 Concatenate . . . 21
3.4 Summary . . . 23

4 Advanced Operations **25**
4.1 Intersect . . . 25
4.2 Compose . . . 30
4.3 Forward and Viterbi . . . 33

5 Differentiation with Automata **39**
5.1 Derivatives . . . 39
5.2 Automatic Differentiation . . . 42

6 Extended Examples **45**
6.1 Counting n-grams . . . 45
6.2 Edit Distance . . . 46
6.3 n-gram Language Model . . . 48
6.4 Automatic Segmentation Criterion . . . 51
6.5 Connectionist Temporal Classification . . . 61

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 $\mathbf{C} = \mathbf{A}\mathbf{B}$. 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:

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

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 $\mathbf{A}$ and $\mathbf{B}$. In graph composition the inner variable is the shared path $\mathbf{r}$.

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].

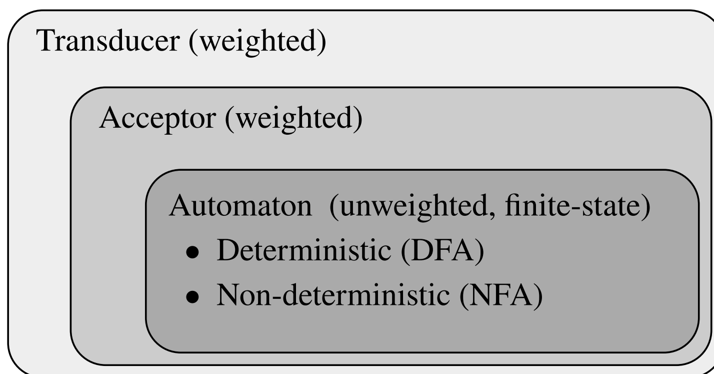


Figure 2.1: A hierarchy of automata classes from general to specific in terms of representation power. Weighted transducers can represent anything that weighted acceptors can represent. Weighted acceptors in turn can represent any unweighted finite-state automata.

2 Acceptors and Transducers

2.1 Automata

The broad class of graphs we are going to look at are finite-state automata. These include deterministic finite-state automata (DFAs) and non-deterministic finite-state automata (NFAs). More specifically we will consider a generalization of DFAs and NFAs called weighted finite-state acceptors (WFSAs). That's a mouthful, so I will just call them *acceptors*. We will also consider a further generalization of an acceptor called a *transducer* (weighted finite-state transducers or WFSTs). Figure 2.1 shows the relation between these three graphs; transducers, acceptors, and automata. Transducers are the most expressive in terms of their representational power, followed by acceptors followed by unweighted automata.

Before we dive into acceptors and transducers, let's introduce some general graph terminology that I will use throughout. In the following graph a *state* or *node* is represented by a circle. The arrows represent connections between two states. We usually refer to these as *arcs* but sometimes also *edges*. The graph is directed since the connections between states are unidirectional arrows. The arcs in a graph can have labels. In figure 2.2 the arc between states 0 and 1 has a label of a. Similarly, the arc between states 1 and 2 has a label of b. The graph is an example of a finite-state automata (FSA) or finite-state machine (FSM), so called because it has a finite number of nodes.

An automata is deterministic if for each state and label pair there is only one outgoing transition which matches that label. An automata is nondeterministic if

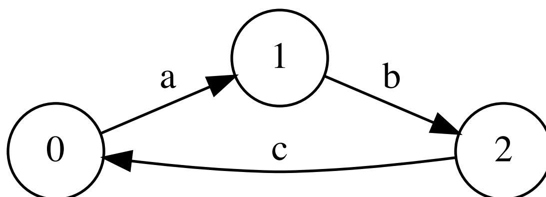
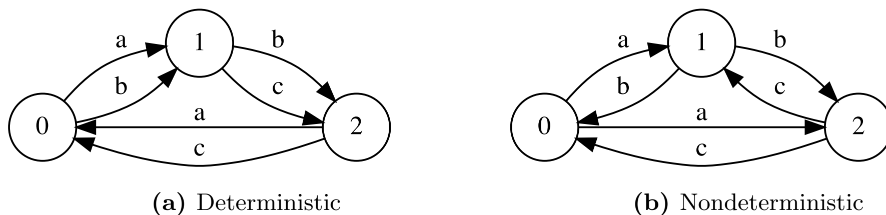


Figure 2.2: An example of a simple finite-state automata.



(a) Deterministic **(b)** Nondeterministic

Figure 2.3: An example of a deterministic automata and a nondeterministic automata. The nondeterministic automata has two arcs leaving state 0 both with label a and two arcs leaving state 2 both with the label c.

multiple transitions leaving a state have the same label. The graphs in figure 2.3 show an example of a deterministic and a nondeterministic automata. In general, acceptors and transducers to can be nondeterministic.

2.2 Acceptors

Let's start by constructing some very basic automata to get a feel for their various properties.

The start state $s = 0$ has a bold circle around it. The accepting state 1 is represented with concentric circles. Each arc has a label and a corresponding weight. So the first arc from state 0 to state 1 with the text $a/0$ means the label is a and the weight is 0. The fact that there is only a single label on each arc means this graph is an *acceptor*. Since it has weights, we say its a weighted acceptor. Since the number of states is finite, some would call it a weighted finite-state acceptor or WFSA. Again, that's a mouthful, so I'll just call these graphs acceptors.

An accepting path in the graph is a sequence of arcs which begin at a start state and end in an accepting state. By concatenating the labels on an accepting path, we get a string which is accepted by the graph. So the string aa is accepted by the

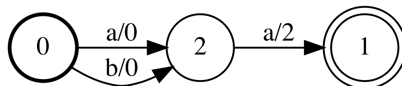


Figure 2.4: An example of a simple acceptor. The label on each arc shows the input label and weight, so the $a/0$ represents a label of a and a weight of 0.

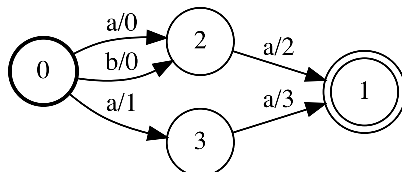


Figure 2.5: An acceptor which has multiple paths for the same sequence, aa.

graph by following the state sequence $0 \rightarrow 2 \rightarrow 1$. The string ba is also accepted by the graph by following the same state sequence but taking the arc with label b when traversing from state 0 to state 1. The language of the acceptor is the set of all strings which are accepted by it. You may also encounter "recognized" used as a synonym for "accepted". Let the variable $\mathcal{A}$ represent the acceptor in figure 2.4. In general, I'll use uppercase script letters to represent graphs. Let $\mathcal{L}(\mathcal{A})$ denote the language of $\mathcal{A}$. In this case $\mathcal{L}(\mathcal{A}) = \{aa, ba\}$.

There are different ways to compute the weight of a string accepted by the graph. The most common is to sum the weights of the arcs on the accepting path for that string. For example the string aa in the graph in figure 2.4 has a weight of $0 + 2 = 2$. Another option would be to multiply the weights. These two options correspond to interpreting the weights as either log probabilities or probabilities. We'll have more to say about this later.

The graph in figure 2.5 accepts the same sequence by multiple paths.

The string aa is accepted along the state sequence $0 \rightarrow 2 \rightarrow 1$ and along the state sequence $0 \rightarrow 3 \rightarrow 1$. In this case, to compute the score of aa we need to consider both paths. Again we have a couple of options here. The most common approach is to *log-sum-exp* the individual path scores. Again this corresponds to interpreting the path scores as log probabilities. We'll use $\text{LSE}(s_1, s_2)$ to denote the *log-sum-exp* of the two scores s_1 and s_2:

$$\text{LSE}(s_1, s_2) = \log\left(e^{s_1} + e^{s_2}\right). \tag{3}$$

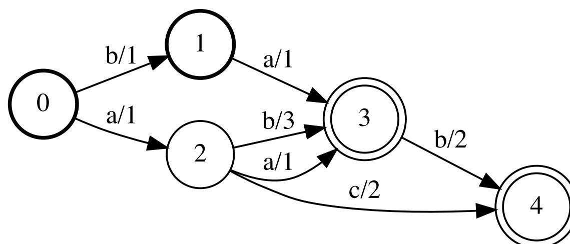


Figure 2.6: An acceptor with multiple start states (0 and 1) and multiple accept states (3 and 4).

So the overall weight for the string aa in the graph in figure 2.5 is given by:

$$\log\left(e^{0+2} + e^{1+3}\right) = 4.13.$$

Acceptors can have multiple start states and multiple accept states. In the graph in figure 2.6, the states 0 and 1 are both start states, and the states 3 and 4 are both accept states.

It turns out that allowing multiple start or accept states does not increase the expressive power of the graph. With ϵ transitions (which we will discuss soon), one can convert any graph with multiple start states and multiple accept states into an equivalent graph with a single start state and a single accept state.

Note also that start states can have incoming arcs (as in state 1) and accept states can have outgoing arcs, as in state 3.

Example 2.1. Compute the score of the string ab in figure 2.6.

The two state sequences which accept the string ab are the states $0 \rightarrow 2 \rightarrow 3$ and $1 \rightarrow 3 \rightarrow 4$. The overall score is given by:

$$\log(e^{1+3} + e^{1+2}) = 4.31.$$

■

Graphs can also have self-loops and cycles. For example, the graph in figure 2.7 has a self-loop on the state 0 and a cycle following the state sequence $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$.

The language of a graph with cycles and self-loops contains infinitely many strings. For example, the language of the graph in figure 2.7 includes any string that starts

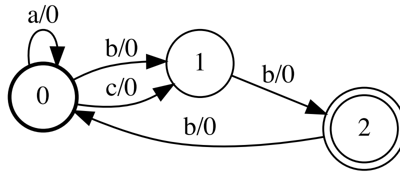


Figure 2.7: A graph with a self-loop on the state 0 and a cycle from $0 \to 1 \to 2 \to 0$.

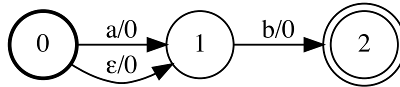


Figure 2.8: An acceptor with an ϵ transition on the second arc between state 0 and 1.

with zero or more as and ends in bb. As a regular expression we write this as a^*bb where the * denotes zero or more as.

The ϵ symbols has a special meaning when it is the label on an arc. Any arc with an ϵ label can be traversed without consuming an input token in the string. So the graph in figure 2.8 accepts the string ab, but it also accepts the string b because we can traverse from state 0 to state 1 without consuming an input.

As it turns out, any graph with ϵ-transitions can be converted to an equivalent graph without ϵ transitions. However, this usually comes at a large cost in the size of the graph. Complex languages can be represented by much more compact graphs with the use of ϵ-transitions.

Example 2.2. Convert the graph in figure 2.6 which has multiple start and accept states to an equivalent graph with only a single start and accept state using ϵ transitions.

The graph in figure 2.9 is the equivalent graph with a single start state and a single accept state.

The construction works by creating a new start state and connecting it to the old start states with ϵ transitions with a weight of 0. The old start nodes are regular internal nodes in this new graph. Similarly the old accept states are now regular states and they connect to the new accept state with ϵ transitions with a weight of 0. ■

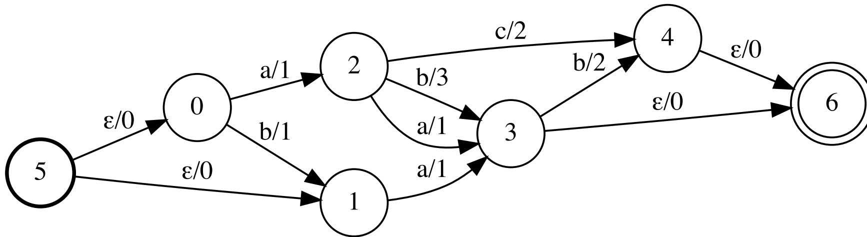


Figure 2.9: The equivalent graph using only a single start state and accept state to the graph in figure 2.6 which has multiple start and accept states.

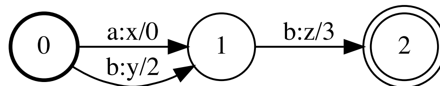


Figure 2.10: An example of a simple transducer. The label on each arc shows the input label, the output label, and the weight. So $a:x/0$ represents an input label of a, and output label of x, and a weight of 0.

2.3 Transducers

A *transducer* maps input strings to output strings. Transducers are a generalization of acceptors. Every acceptor is a transducer, but not every transducer is an acceptor. Let's look at a few example transducers to understand how they work.

The arc labels distinguish an acceptor from a transducer. A transducer has both an input and output arc label. The arc labels are of the form $a:x/0$ where a is the input label x is the output label and 0 is the weight. An acceptor can be represented as a transducer where the input and output labels on every arc are identical.

Instead of saying that a transducer accepts a given string, we say that it *transduces* one string to another. The graph in figure 2.10 transduces the string ab to the string xz and the string bb to the string yz. The weight of a transduced pair is computed in the same way as in an acceptor. The scores of the individual arcs on the path are summed. The path scores are combined with *log-sum-exp*. So the weight of the transduced pair (ab, xz) in the graph in figure 2.10 is $0 + 3 = 3$.

We have to generalize concept of the language from an acceptor to a transducer. I'll call this generalization the transduced set. Since it will always be clear from context if the graph is an acceptor or transducer, I'll use the same symbol $\mathcal{L}$ to represent the transduced set. If $\mathcal{T}$ is a transducer, then $\mathcal{L}(\mathcal{T})$ is the set of pairs

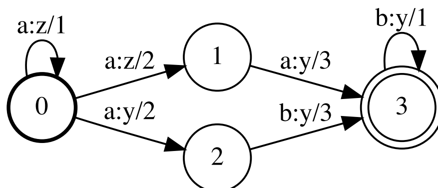


Figure 2.11: An example transducer in which the sequence aab is transduced to the sequence zyy on multiple paths.

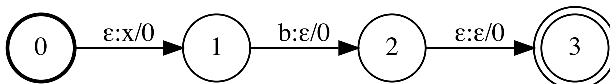


Figure 2.12: A transducer with ϵ transitions. The ϵ can be just the input label, just the output label, or both the input and output label.

of strings transduced by $\mathcal{T}$. More formally, a pair of strings $(\mathbf{x}, \mathbf{y}) \in \mathcal{L}(\mathcal{T})$ if $\mathcal{T}$ transduces $\mathbf{x}$ to $\mathbf{y}$.

Example 2.3. Compute the score of the transduced pair (aab, zyy) in the graph in figure 2.11.

The two paths which transduce aab to zyy are following the state sequence $0 \rightarrow 1 \rightarrow 3 \rightarrow 3$ and $0 \rightarrow 0 \rightarrow 2 \rightarrow 3$. The score of the first path is 6 and the score of the second path is 6. So the overall score is:

$$\log\left(e^6 + e^6\right) = 6.69.$$

■

Transducers can also have ϵ transitions. The ϵ can be either the input label on an arc, the output label on an arc, or both. When the ϵ is the input label on an arc, it means we can traverse that arc without consuming an input token, but we still output the arc's corresponding output label. When the ϵ is the output label, the opposite is true. The input is consumed but no output is produced. And when the ϵ is both the input and the output label, the arc can be traversed without consuming an input or producing an output.

In the graph in figure 2.12, the string b gets transduced to the string x. On the first arc between states 0 and 1, we output an x without consuming any token. On the second arc between states 1 and 2, a b is consumed without outputting

any new token. Finally, on the arc between states 2 and 3 we neither consume nor output a token.

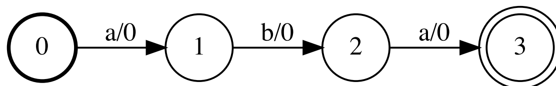


Figure 3.1: An example of a simple acceptor with language $\{aba\}$, in other words $\mathcal{L}(\mathcal{A}) = \{aba\}$.

3 Basic Operations

An operation on a transducer (or acceptor) takes one or more transducers as input and outputs a transducer. You can think of these operations as functions on graphs. As a reminder, I'll use uppercase script letters to represent graphs, so $\mathcal{A}$ for example can represent a graph. Functions will be denoted by lower case variables. So $f(\mathcal{A})$ is a function which takes as input a single graph and outputs a graph.

3.1 Closure

The closure, sometimes called the Kleene star, is a unary function (takes a single input) which can operate on either an acceptor or transducer. If the sequence $\mathbf{x}$ is accepted by $\mathcal{A}$, then zero or more copies of $\mathbf{x}$ are accepted by the closure of $\mathcal{A}$. More formally, if the language of an acceptor is $\mathcal{L}(\mathcal{A})$, then the language of the closure of $\mathcal{A}$ is $\{\mathbf{x}^n \mid \mathbf{x} \in \mathcal{L}(\mathcal{A}), \;\; n = 0, 1, \ldots, \}$. The notation $\mathbf{x}^n$ means $\mathbf{x}$ concatenated n times. So $\mathbf{x}^2$ is $\mathbf{xx}$ and $\mathbf{x}^0$ is the empty string. Usually the closure of an acceptor is denoted by *, as in $\mathcal{A}^*$. This is the same notation used in regular expressions.

The closure of a graph is easy to construct with the use of ϵ transitions. The language of the graph in figure 3.1 is the string *aba*.

The closure of the graph needs to accept an arbitrary number of copies of *aba* including the empty string. To accept the empty string we make the start state an accept state as well. To accept one or more copies of *aba* we simply wire up the old accept states to the new start state with ϵ transitions.

The closure of the graph in figure 3.1 is shown in figure 3.2.

Example 3.1. You might notice that state 4 in the graph in figure 3.2 is not necessary. Consider an alternate construction for computing the closure of a graph. We could have made the state 0 into an accept state and connected state 3 to state 0 with an ϵ transition, as in the graph in figure 3.3.

For the graph in figure 3.1, this alternate construction works and requires fewer

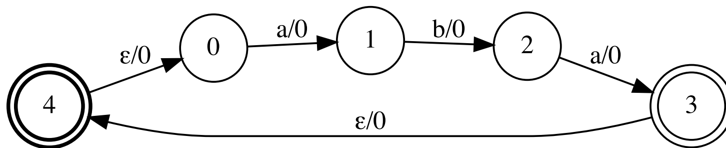


Figure 3.2: The closure $\mathcal{A}^*$ of the graph in figure 3.1. The language of the graph is $\{\epsilon, aba, abaaba, \ldots\}$.

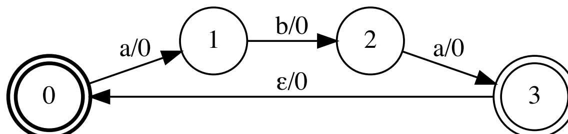


Figure 3.3: The closure $\mathcal{A}^*$ of the graph in figure 3.1 using an alternate, simpler construction which connects the accept state to the original start state with an ϵ transition. This construction does not work for every case.

states and arcs. In the general case, this construction turns every start state into an accept state instead of adding a new start state. Give an example where this doesn't work? In other words, give an example where the graph from this modified construction is not the correct closure of the original graph.

An example for which the alternate construction does not work is shown in the graph in figure 3.4. The language of the graph is $a^n b$ (any number of a's followed by a b) and the closure is $(a^n b)^*$, or any sequence that ends with b.

If we follow the modified construction for the closure, as in the graph in figure 3.5, then the language would incorrectly include sequences that do not end with b such as a^*. The graph following the correct construction of the closure is in figure 3.6.

■

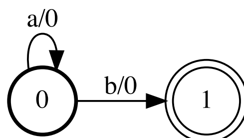


Figure 3.4: An acceptor for which the alternate construction for the closure does not yield the correct result.

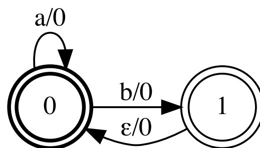


Figure 3.5: The alternate construction which incorrectly computes the closure of the graph in figure 3.4.

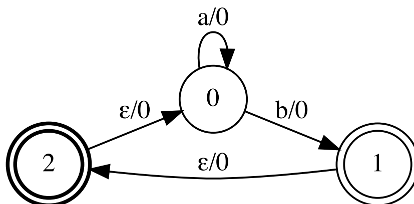


Figure 3.6: The correct closure of the graph in figure 3.4 which has the language $(a^n b)^*$, which is any sequence which ends with b.

3.2 Union

The union takes as input two or more graphs and produces a new graph. The language of the resultant graph is the union of the languages of the input graphs. More formally let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be n graphs. The language of the union graph is given by $\{x \mid x \in \mathcal{A}_i \text{ for some } i = 1, \ldots, n\}$. I'll occasionally use the $+$ sign to denote the union, as in $\mathcal{U} = \mathcal{A}_1 + \mathcal{A}_2$.

Since we let a graph have multiple start states and multiple accept states, the union is easy to construct. A state in the union graph is a start state if it was a start state in one of the original graphs. A state in the union graph is an accept state if it was an accept state in one of the original graphs.

Consider the three graphs in figure 3.7 with languages $\{ab, aba, abaa, \ldots\}$, $\{ba\}$, and $\{ac\}$ respectively.

Notice in the union graph in figure 3.8 the only visual distinction from the individual graphs is that the states are numbered consecutively from 0 to 8 indicating a single graph with nine states instead of three individual graphs. The language of the union graph is $\{ab, aba, abaa, \ldots\} \cup \{ba\} \cup \{ac\}$.

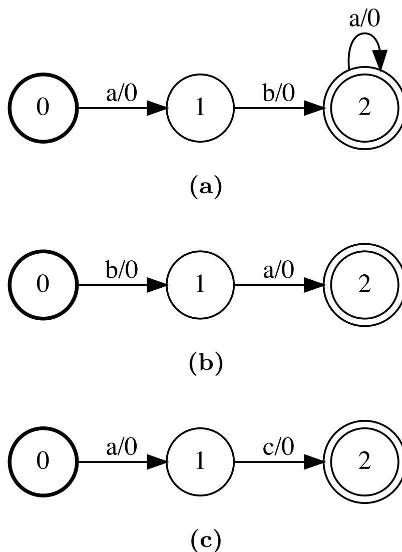


Figure 3.7: Three acceptors with languages $\{ab, aba, abaa, \ldots\}$, $\{ba\}$, and $\{ac\}$ from top to bottom, respectively.

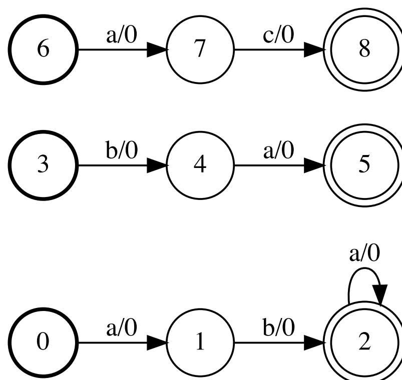


Figure 3.8: The union of the three acceptors in figure 3.7 with language $\{ab, aba, abaa, \ldots\} \cup \{ba\} \cup \{ac\}$.

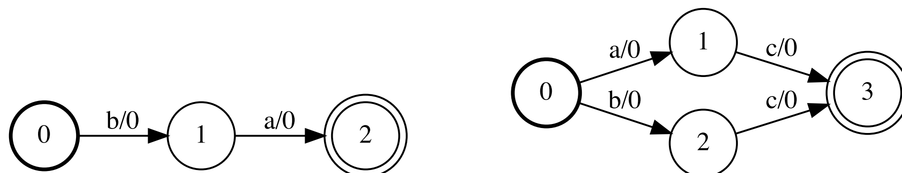


Figure 3.9: The acceptor on left has language $\{ba\}$ and the acceptor on the right has language $\{ac, bc\}$.

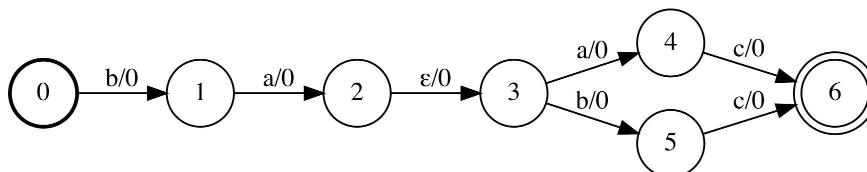


Figure 3.10: The graph is the concatenation of the two graphs in figure 3.9 and has the language $\{baac, babc\}$.

3.3 Concatenate

Like union, concatenate produces a new graph given two or more graphs as input. The language of the concatenated graph is the set of strings which can be formed by any concatenation of strings from the individual graph. Concatenate is not commutative, the order of the input graphs matters. More formally the language of the concatenated graph is given by $\{\mathbf{x}_1 \dots \mathbf{x}_n \mid \mathbf{x}_1 \in \mathcal{L}(\mathcal{A}_1), \dots, \mathbf{x}_n \in \mathcal{L}(\mathcal{A}_n)\}$. Occasionally, I will denote concatenation by placing the graphs side-by-side. So $\mathcal{A}_1\mathcal{A}_2$ represents the concatenation of $\mathcal{A}_1$ and $\mathcal{A}_2$.

The concatenated graph can be constructed from the original input graphs by connecting the accept states of one graph to the start states of the next. Assume we are concatenating $\mathcal{A}_1, \dots, \mathcal{A}_n$. The start states of the concatenated graph are the start states of the first graph, $\mathcal{A}_1$. The accept states of the concatenated graph are the accept states of the last graph, $\mathcal{A}_n$. For any two graph $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$, we connect each accept state of $\mathcal{A}_i$ to each start state of $\mathcal{A}_{i+1}$ with an ϵ transition.

As an example, consider the two graphs in figure 3.9. The concatenated graph is in figure 3.10 and has the language $\{baac, babc\}$.

Example 3.2. What is the identity graph for the concatenation function? The identity in a binary operation is the value which when used in the operation leaves the second input unchanged. In multiplication this would be 1 since $c * 1 = c$ for any real value c.

(a) Identity **(b)** Annihilator

Figure 3.11: The identity and the annihilator for the concatenate operation. The language of the identity graph is the empty string $\{\epsilon\}$. The language of the annihilator graph is the empty set $\{\}$.

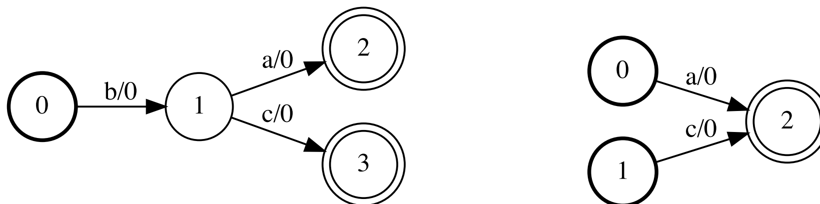


Figure 3.12: The acceptor on the right has the language $\{ba, bc\}$, the acceptor on the left has the language $\{a, c\}$.

What is the equivalent of the annihilator graph in the concatenation function? The annihilator in a binary operation is the value such that the operation with the annihilator always returns the annihilator. For multiplication 0 is the annihilator since $c * 0 = 0$ for any real value c.

The graph which accepts the empty string is the identity. The graph which does not accept any strings is the annihilator. See the figure 3.11 for an example of these two graphs.

The identity graph is a single node which is both a start and accept state. The language of the identity graph is the empty string. The annihilator graph is a single non accepting state. The language of the annihilator graph is the empty set. Note the subtle distinction between the language that contains the empty string and the language that is the empty set. The former can be written as $\{\epsilon\}$ whereas the latter is $\{\}$ (also commonly denoted by $\varnothing$). ■

Example 3.3. Construct the concatenation of the two graphs in figure 3.12.

The concatenated graph is in figure 3.13.

■

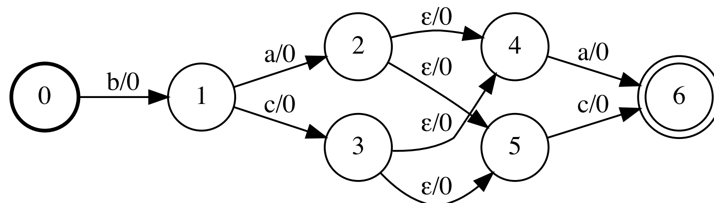


Figure 3.13: The concatenation of the two graphs in figure 3.12 has the language $\{baa, bac, bca, bcc\}$.

Example 3.4. Suppose we have a list of graphs to concatenate $\mathcal{A}_1, \ldots, \mathcal{A}_n$ where the i-th graph has s_i start states and a_i accept states. How many new arcs will the concatenated graph require?

For each consecutive pair of graphs $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$, we need to add $a_i * s_{i+1}$ connecting arcs in the concatenated graph. So the total number of additional arcs is:

$$\sum_{i=1}^{n-1} a_i * s_{i+1}.$$

■

3.4 Summary

We've seen three basic operations so far:

- **Closure:** The closed graph accepts any string in the input graph repeated zero or more times. The closure of a graph $\mathcal{A}$ is denoted $\mathcal{A}^*$.
- **Union:** The union graph accepts any string from any of the input graphs. The union of two graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ is denoted $\mathcal{A}_1 + \mathcal{A}_2$.
- **Concatenate:** The concatenated graph accepts any string which can be formed by concatenating strings (respecting order) from the input graphs. The concatenation of two graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ is denoted $\mathcal{A}_1\mathcal{A}_2$.

Example 3.5. Assume you are given the following individual graphs $\mathcal{A}_a$, $\mathcal{A}_b$, and $\mathcal{A}_c$, which recognize a, b, and c respectively, as in figure 3.14. Using only closure, union, and concatenate, construct the graph which recognizes any number of repeats of the strings aa, bb, and cc. For example $aabb$ and $bbaacc$ are in the language but b and $ccaab$ are not.

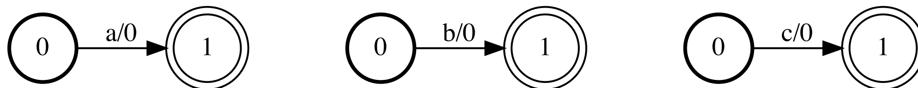


Figure 3.14: The three individual automata with languages $\{a\}$, $\{b\}$, and $\{c\}$ from left to right, respectively.

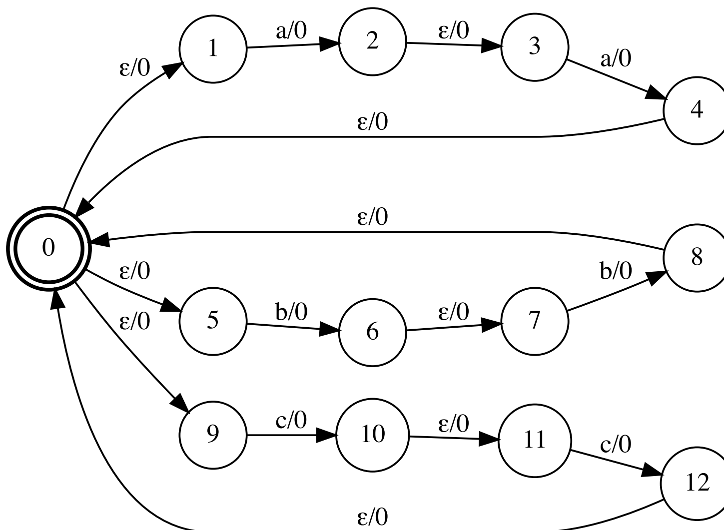


Figure 3.15: The even numbered repeats graph constructed from the individual token graphs using the operations $\mathcal{A} = (\mathcal{A}_a\mathcal{A}_a + \mathcal{A}_b\mathcal{A}_b + \mathcal{A}_c\mathcal{A}_c)^*$.

First concatenate the individual graphs with themselves to get graphs which recognize aa, bb, and cc. Then take the union of the three concatenated graphs followed by the closure. The resulting graph is shown in figure 3.15. The equation to compute the desired graph is $\mathcal{A} = (\mathcal{A}_a\mathcal{A}_a + \mathcal{A}_b\mathcal{A}_b + \mathcal{A}_c\mathcal{A}_c)^*$.

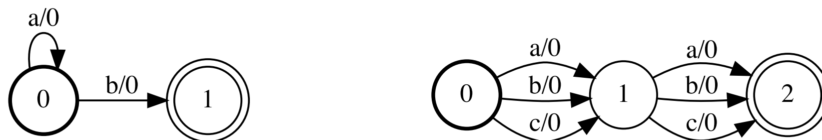


Figure 4.1: The acceptor on the left has a language of a^*b and the acceptor on the right accepts all nine sequences of length 2. The only sequence accepted by both, and hence is in the intersection, is the string *ab*.

4 Advanced Operations

4.1 Intersect

The language of the intersection of two acceptors is the set of strings which is accepted by both of them. The score of a path in the intersected graph is the sum of the score of the path in each of the input graphs. More formally the language of the intersected graph is given by $\{\mathbf{x} \mid \mathbf{x} \in \mathcal{L}(\mathcal{A}_1) \;\wedge\; \mathbf{x} \in \mathcal{L}(\mathcal{A}_2)\}$.

The analogous operation for transducers is the composition, which I will discuss in more detail in the next section. In most machine learning applications, intersect and compose tend to be the primary operations used to compute more complex graphs from simpler input graphs. Thus gaining a deeper understanding of these two operations is worth the time.

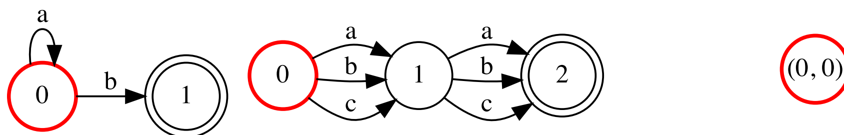
We'll use the two graphs in figure 4.1 to illustrate a general algorithm for computing the intersection. In this case, the intersected graph is easy to see by inspecting the two graphs. The only string which is recognized by both graphs is the string *ab*, hence the intersected graph is the graph which recognizes *ab*.

The general algorithm relies on a queue to explore pairs of states from the two input graphs. Pseudocode is given in algorithm 4.1.

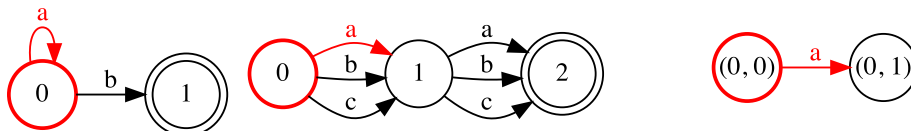
Some of the steps of the intersect algorithm on the two input graphs in figure 4.1 are shown in the sequence of graphs in figure 4.2. The states explored at each step are highlighted in red. The intersected graph is the third graph on the right which is constructed over the steps of the algorithm.

The pseudocode in algorithm 4.1 does not handle ϵ transitions. The challenge with including ϵ transitions in intersect and compose is discussed in section 4.2.1.

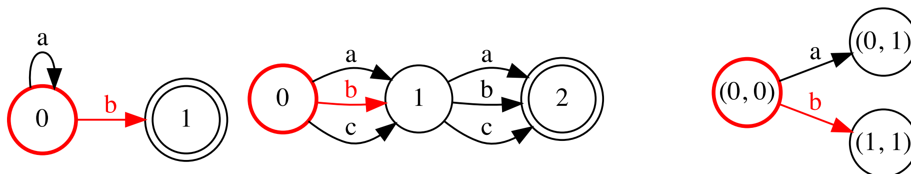
Another potential issue with the algorithm 4.1is that the output graph it constructs is not *trim*. An automata is trim if every state in the graph is part of a path which starts in a start state and terminates in an accept state. In short, the graph



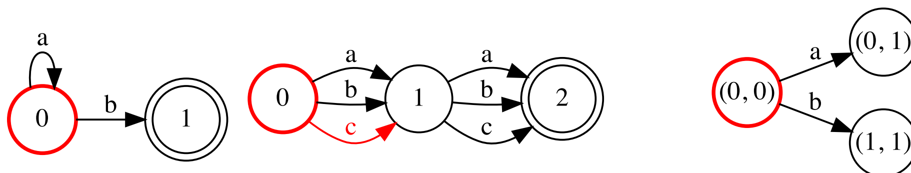
(a) The starts states in the input graphs are 0 and 0. So we add the start state $(0, 0)$ to the intersected graph and to the queue to be explored.



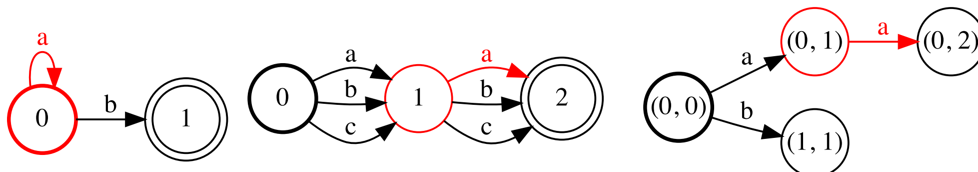
(b) The first pair of outgoing arcs match on the label a. This means the downstream state $(0, 1)$ is reachable in the intersected graph. So we add $(0, 1)$ to the intersected graph and to the queue to be explored.



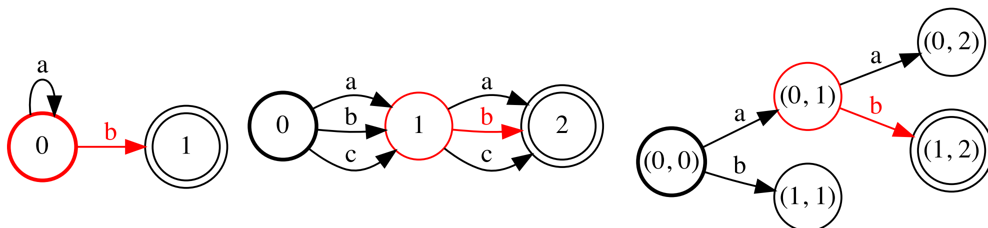
(c) The next matching pair of outgoing arcs match on the label b. Also, we haven't visited the downstream state $(1, 1)$ yet. So we add $(1, 1)$ to the intersected graph and to the queue to be explored.



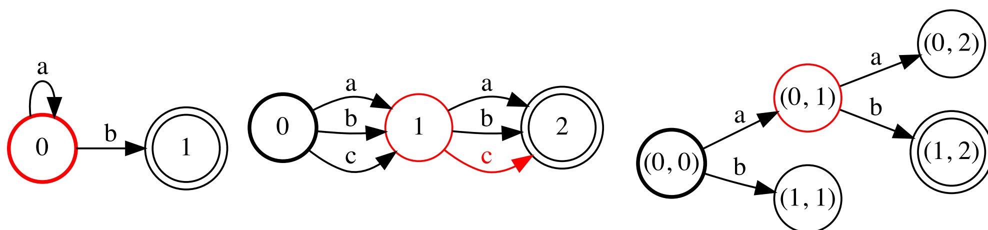
(d) The arc in the second input graph with label c does not have a matching arc from state 0 in the first input graph, so it is ignored.



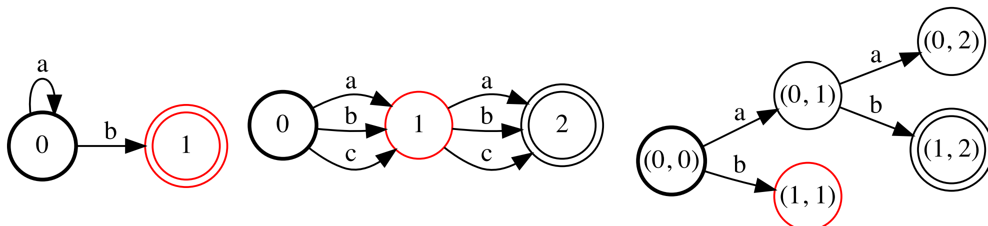
(e) At this point we have considered all the matching outgoing arcs from $(0, 0)$ so we now move on to the next state pair in the queue, $(0, 1)$. From $(0, 1)$, a pair of outgoing arcs match on the label a. The downstream state $(0, 2)$ is new so we add it to the intersected graph and to the queue.



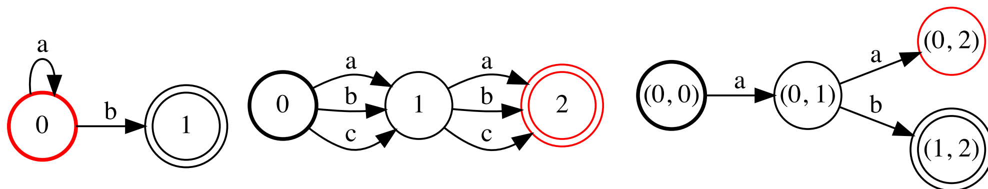
(f) The next pair of matching outgoing arcs match on the label b and lead to the state pair $(1, 2)$. We add $(1, 2)$ to the queue and to the intersected graph. Also, since 1 is an accept state in the first graph and 2 is an accept state in the second graph, $(1, 2)$ is an accept state in the intersected graph.



(g) Again, the arc in the second input graph with label c does not have a matching arc from state 0 in the first input graph, so it is ignored.



(h) The next state pair in the queue is $(1, 1)$. There are no arcs leaving state 1 in the first input graph, and $(1, 1)$ is not an accept state in the intersected graph, so it is a dead end. We can remove $(1, 1)$ and its incoming arcs from the intersected graph.



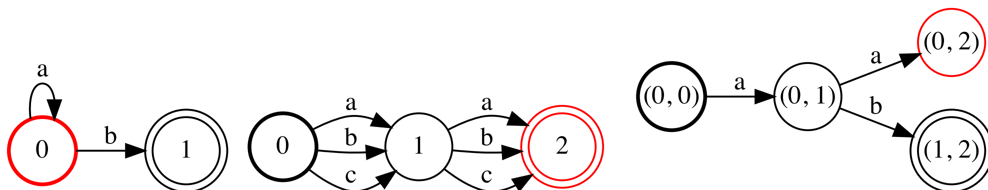
(i) The next state pair in the queue is $(0, 2)$. Again, there are no matching arcs leaving this state pair, and $(0, 2)$ is not an accept state in the intersected graph, hence it is a dead end. We can also remove $(0, 2)$ and its incoming arcs from the intersected graph.

Algorithm 4.1 Intersect

1: **Input**: Acceptors $\mathcal{A}_1$ and $\mathcal{A}_2$
2: Initialize the queue Q and the intersected graph $\mathcal{I}$.
3: **for** s_1 and s_2 in all start state pairs of $\mathcal{A}_1$ and $\mathcal{A}_2$ **do**
4: Add (s_1, s_2) to Q and as a start state in $\mathcal{I}$.
5: **if** s_1 and s_2 are accept states **then**
6: Make (s_1, s_2) an accept state in $\mathcal{I}$.
7: **end if**
8: **end for**
9: **while** Q is not empty **do**
10: Remove the next state pair (u_1, u_2) from Q.
11: **for** all arcs pairs a_1 and a_2 leaving u_1 and u_2 with matching labels **do**
12: Get destination states v_1 of a_1 and v_2 of a_2.
13: **if** not yet seen (v_1, v_2) **then**
14: Add (v_1, v_2) as a state to $\mathcal{I}$ and to Q.
15: **if** v_1 and v_2 are accept states **then**
16: Make (v_1, v_2) an accept state in $\mathcal{I}$.
17: **end if**
18: **end if**
19: Get the label ℓ of a_1.
20: Get the weights w_1 of a_1 and w_2 of a_2.
21: Add an arc from (u_1, u_2) to (v_1, v_2) with label ℓ and weight $w_1 + w_2$.
22: **end for**
23: **end while**
24: **Return**: The intersected graph $\mathcal{I}$.

does not have any useless states. In the standard terminology, a state is said to be *accessible* if it can be reached from a start state and *coaccessible* if an accept state can be reached from it. A graph is trim if every state is both accessible and coaccessible.

Every state in the intersected graph $\mathcal{I}$ will be accessible, but not necessarily coaccessible, so $\mathcal{I}$ may not be trim. However, the graph will still be correct, so this is primarily an issue of representation size. With some additional work the constructed graph can be kept trim during the operation of the algorithm. Another alternative is to construct a trim graph from the non-trim graph as a post-processing step. Note, in figure 4.2 showing some steps of the intersect algorithm, we removed dead-end states in the intersected graph for clarity; however, algorithm 4.1 does not do this.



(j) The next state pair in the queue is $(1, 2)$. There are no matching arcs for this state pair, but it is an accept state in the intersected graph, so we have to keep it. At this point the queue is empty. The algorithm terminates, and we are left with the complete intersected graph.

Figure 4.2: An illustration of some of the steps (a-j) taken in the intersect algorithm. The two input graphs are on the left and the middle and the construction of the intersected graph is shown on the right.

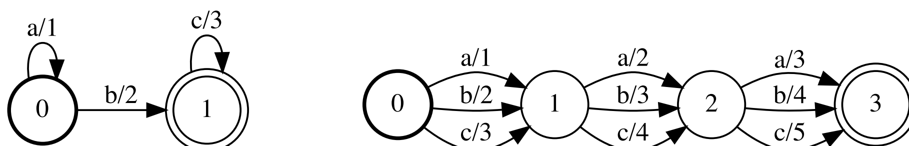


Figure 4.3: The automata on the left recognizes a^*bc^*, the automata on the right recognizes all three letter combinations of the alphabet $\{a, b, c\}$.

Example 4.1. Compute the intersection of the two graphs in figure 4.3. Make sure to update the arc weights correctly in the intersected graph.

The intersection is given in figure 4.4. The intersected graph accepts the sequences *aab*, *abc*, and *bcc* which are the only sequences accepted by both inputs.

■

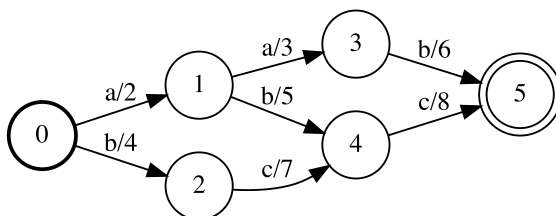


Figure 4.4: The intersection of the graphs in figure 4.3 accepts the strings *aab*, *abc*, and *bcc*.

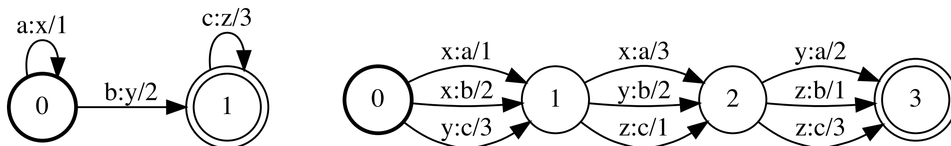


Figure 4.5: Two transducers for which we would like to compute the composition.

4.2 Compose

The composition is a straightforward generalization of the intersection from acceptors to transducers. Assume the first input graph transduces the sequence **x** to the string **y** and the second graph transduces **y** to **z**. Then the composed graph transduces **x** to **z**.

From an implementation standpoint the compose and intersect algorithms are almost identical. The two minor differences are 1) the way that labels are matched in the input graphs and 2) the labels of the new arcs in the composed graph. Arcs in a transducer have both input and output labels. We match the output arc label from the first graph to the input arc label from the second graph. This means that compose, unlike intersect, is not commutative, since the order of the two graphs makes a difference.

The input label of a new arc in the composed graph is the input label of the corresponding arc in the first input graph. The output label of a new arc in the composed graph is the output label of the corresponding arc in the second input graph. For example, assume we have two arcs, the first with label $x_1 : x_2$, and the second with label $y_1 : y_2$. If $x_2 = y_1$, then the arcs are considered a match. The new arc will have the label $x_1 : y_2$.

Think of matrix multiplication as an analogy or mnemonic device. When multiplying two matrices they have to match on the inner dimension, and the dimensions of the output matrix are the outer dimensions. In the same way the inner labels of the two arcs must match and the resulting arc labels are the outer labels of the two input arcs.

Example 4.2. Compute the composition of the two graphs in figure 4.5.

The composition is given in figure 4.6. As an example, the first input graph transduces *abc* to xyz and the second input transduces xyz to *abb*, *abc*, *bbb*, and *bbc*. Thus, the composed graph should transduce *abc* to all four of *abb*, *abc*, *bbb*, and *bbc*. You can verify this in the graph in figure 4.6.

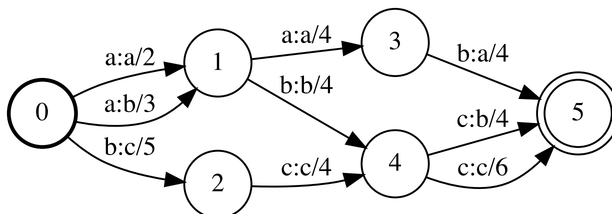


Figure 4.6: The composition of the two graphs from figure 4.5.

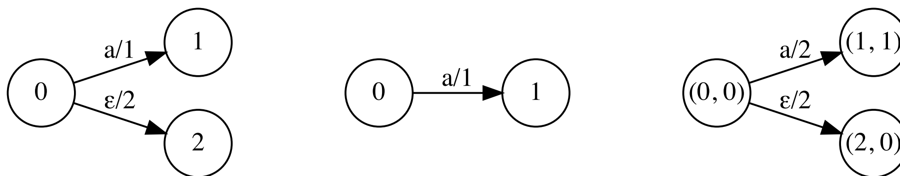


Figure 4.7: The graph on the left has an ϵ transition. Computing the intersection of the left and middle graph results in the graph on the right. The ϵ transition can be followed without consuming an input in the middle graph.

■

4.2.1 Intersect and Compose with ϵ

The basic implementation of intersect and compose we have discussed so far doesn't extend to ϵ transitions. Allowing ϵ transitions in these algorithms makes them more complicated. In this section I will illustrate the challenges with a naive approach and sketch at a high-level how to actually account for ϵ transitions in the intersect and compose algorithms.

First, consider the simpler case when only the first input graph has ϵ transitions. In this case, whenever we encounter an outgoing ϵ transition from a state in the first graph, we can optionally traverse it without matching a corresponding arc in the second graph.

Consider the sub-graphs in figure 4.7. Suppose we are currently looking for outgoing arcs with matching labels from the state $(0, 0)$. When we find a matching pair, we add the new state to the intersected graph and add a corresponding arc. In the ϵ-free case, we only explore the arc's with label a. The state $(1, 1)$ is added to the intersected graph and the queue to be explored. The state $(0, 0)$ is connected to the state $(1, 1)$ with an arc with label a and weight 2.

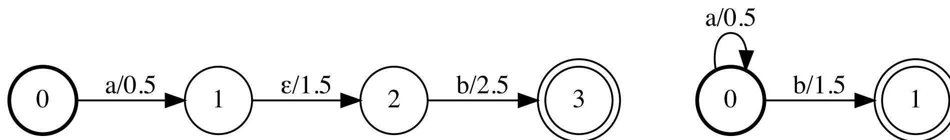


Figure 4.8: An example of two acceptors for which we would like to compute the intersection. The acceptor on the left has an ϵ transition.

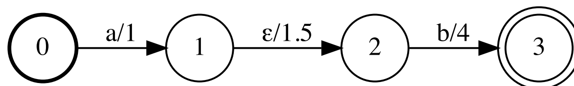


Figure 4.9: The intersection of the two graphs in figure 4.8.

Since state 0 in the first graph has an outgoing ϵ, we can optionally traverse it without traversing any arc in the second graph. In this case, we add the state $(2, 0)$ to the intersected graph and to the queue to be explored. We also add an arc from the state $(0, 0)$ to the state $(2, 0)$ in the intersected graph with a label ϵ and a weight of 2.

Example 4.3. Compute the intersection of the two graphs in figure 4.8.

The intersected graph is in figure 4.9. The ϵ transition is included for clarity, though it could be removed and states 1 and 2 collapsed yielding an equivalent graph.

■

The trickier case to handle is when both graphs have ϵ transitions. If we optionally explore outgoing ϵ arcs in each graph, then we will end up with too many paths in the intersection. Suppose we are given the two graphs in figure 4.10, each of which has an ϵ transition.

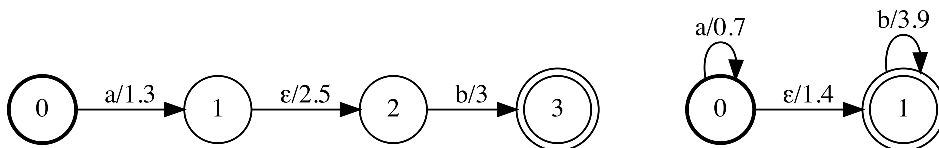


Figure 4.10: Two acceptors, both of which have ϵ transitions.

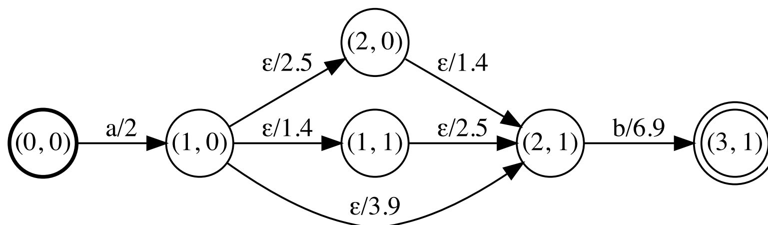


Figure 4.11: An incorrect composition of the two graphs in figure 4.10. Naively following ϵ transitions as we encounter them when computing the intersection results in an incorrect graph. The graph admits too many paths for the sequence ab, and hence the score of that sequence is incorrect.

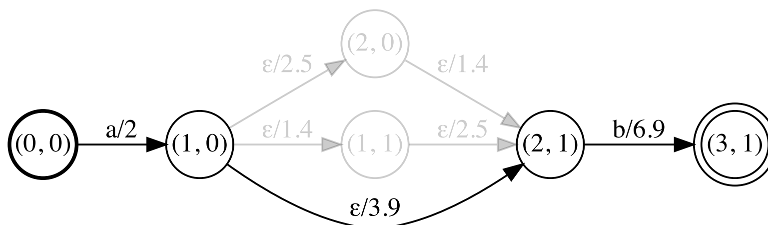


Figure 4.12: Correctly accounting for ϵ transitions when intersecting the two graphs in figure 4.10. Notice that only one of the three resultant paths should be retained in the intersected graph.

If we compute the intersection of the two graphs in figure 4.10, optionally following ϵ transitions as we encounter them, then we end up with the graph in figure 4.11.

The language of this graph is correct. It accepts the string ab which is the only string in the intersection. However, the weight it assigns to the string ab is incorrect. Each individual path has the correct weight, but there are three paths for ab. The final weight will receive three contributions, one from each path, instead of a single contribution from one path. The solution to this problem is to choose only one of the three paths and avoid the inadvertent redundancy. For example we could keep the bottom path and ignore the top two as in the graph in figure 4.12.

4.3 Forward and Viterbi

The forward score and the Viterbi score take a graph as input and return a single scalar result. The *forward score* is the accumulation of the weights of all possible paths from any start state to any accept state in the graph. The weight of the

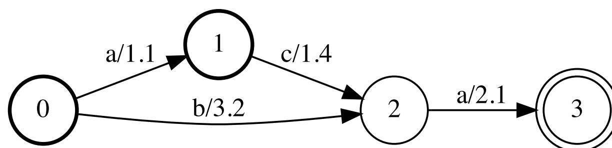


Figure 4.13: An acceptor with three paths from the start states 0 and 1 to the accept state 3.

highest scoring path is the *Viterbi score*, and the path itself is the *Viterbi path*.

> Shortest Distance
>
> In some descriptions of weighted automata, the forward and Viterbi score are introduced as shortest distance algorithms under a respective semiring. The forward score corresponds to the log semiring and the Viterbi score corresponds to the tropical semiring. This is a more general perspective, and useful if you intend to use other semirings. However, we only need the log and tropical semirings in all of the applications we study, so I will restrict the description to the more specific forward and Viterbi score.

Let's start with a couple of examples to show exactly what we are trying to compute, then we will go through a more general algorithm for forward and Viterbi scoring. For the forward and Viterbi score, we will restrict the graphs to be acyclic, meaning no self-loops or cycles. Under certain technical conditions a graph with cycles can admit a computable forward and Viterbi score, but we won't discuss these cases as they don't come up often in machine-learning applications.

The graph in figure 4.13 has three possible paths from the start states to the accept state. The paths and their scores are:

- State sequence $0 \to 1 \to 2 \to 3$ with score 4.6
- State sequence $0 \to 2 \to 3$ with score 5.3
- State sequence $1 \to 2 \to 3$ with score 3.5

The Viterbi score is the maximum over the individual path scores, in this case $\max\{4.6, 5.3, 3.5\} = 5.3$. The Viterbi path is the sequence of labels which correspond to the arcs contributing to the Viterbi score. We represent the Viterbi path as a simple linear graph as in figure 4.14. Note the Viterbi path may not be unique–multiple paths could all attain the Viterbi score. The forward score is the *log-sum-exp* over all the path scores, in this case $\text{LSE}(4.6, 5.3, 3.5) = 5.81$. The

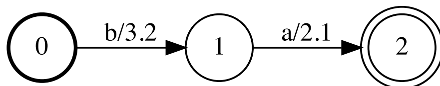


Figure 4.14: The Viterbi path for the graph in figure 4.13. The score of the Viterbi path is the Viterbi score, in this case 5.3.

forward score will always be larger than the Viterbi score. However, the two will converge as the difference between Viterbi score and the second highest scoring path increases.

We computed the forward and Viterbi score by listing all possible paths, computing their individual weights, and then accumulating either with the max or the LSE operations. This approach won't scale to larger graphs because the number of paths can grow combinatorially. Instead, we will use a much more efficient dynamic programming algorithm which works for both the forward and Viterbi score.

The dynamic programming algorithm relies on the following recursions. Consider a state v and let e_i for $i = 1, \ldots, k$ be the set of arcs for which v is the destination node. For a given arc e we let $\text{source}(e)$ denote the source node for that arc. The score of all paths which start at a start state and terminate at node v can be constructed from the score of all paths which start at a start state and terminate at the state $\text{source}(e_i)$ and the arc weight $w(e_i)$ for $i = 1 \ldots, k$. For the Viterbi score, the recursion is:

$$s_v = \max_{i=1}^{k} \left(s_{\text{source}(e_i)} + w(e_i) \right),$$

where s_v is the score of all paths starting at a start state terminating at state v, and $s_{\text{source}(e_i)}$ is the score of all paths starting at a start state terminating at state $\text{source}(e_i)$. The recursion is shown graphically in figure 4.15. In figure 4.15, the Viterbi score of state 3 is the maximum over the weight plus the source node score for all incoming arcs.

The overall Viterbi score is the max of the Viterbi scores over the accept states, $\max_a s_a$ where a is an accept state.

The forward score uses the exact same recursion but with an LSE in place of the max:

$$s_v = \operatorname*{LSE}_{i=1}^{k} \left(s_{\text{source}(e_i)} + w(e_i) \right),$$

and the final score is $\text{LSE}_a\, s_a$.

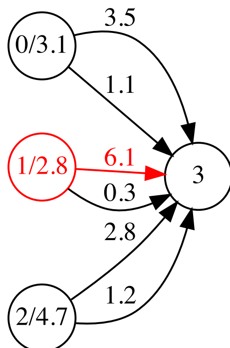


Figure 4.15: A graphical depiction of the recursion for computing the Viterbi score.

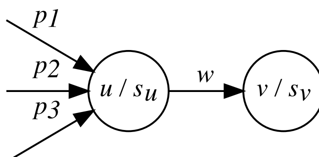


Figure 4.16: The recursion in computing the Viterbi and forward score works because the score s_v at state v can be computed from the weight w and the score s_u at state u.

For both the forward and Viterbi score, the recursion works because the LSE and max operations admit a simple decomposition of the score for all paths terminating at a given state. Suppose, as in the graph in figure 4.16, we have a state v for which we want to compute the score s_v. Suppose also that v has only one incoming arc from state u. Three paths from a start state terminate at state u with the given scores p_1, p_2, and p_3. We can extend each of the three paths from u to v by adding the arc between them, so there are three paths terminating at v as well.

First, suppose we want to compute the Viterbi score at v. The Viterbi score is the maximum of the weights of all three paths terminating at v, namely $s_v = \max\{w + p_1, w + p_2, w + p_3\}$. We can also compute the Viterbi score s_v from the Viterbi score, s_u, of paths terminating at u. In this case $s_v = w + s_u$. This is the recursion we used above but for simplicity with only one incoming arc to v.

The two ways of computing s_v are equivalent:

$$\begin{aligned} s_v &= w + s_u \\ &= w + \max\{p_1, p_2, p_3\} \\ &= \max\{w + p_1, w + p_2, w + p_3\} \\ &= s_v. \end{aligned}$$

The same decomposition works for the forward score and the LSE operation:

$$\begin{aligned} s_v &= w + s_u \\ &= w + \log\left(e^{p_1} + e^{p_2} + e^{p_3}\right) \\ &= \log e^w + \log\left(e^{p_1} + e^{p_2} + e^{p_3}\right) \\ &= \log e^w \left(e^{p_1} + e^{p_2} + e^{p_3}\right) \\ &= \log\left(e^{w+p_1} + e^{w+p_2} + e^{w+p_3}\right) \\ &= s_v \end{aligned}$$

For simplicity, we assume only three paths terminating at u and one arc incoming to v. The argument is easily extended to an arbitrary number of paths and arcs.

4.3.1 Viterbi Path

A Viterbi path is a path for which the Viterbi score is attained. We can compute one of the Viterbi paths with a straightforward extension of the Viterbi scoring algorithm. At each state when we compute the score, we also maintain a back-pointer to the arc which resulted in the maximum score. When the algorithm terminates, we can trace the back-pointers to the start state and extract the Viterbi path.

An example of this can be seen in the graph in figure 4.17.

Each state in the graph is labeled with the Viterbi score over all paths terminating at that state. The red arcs are the arcs which result in the maximum score for the state that they point to. In order to compute the Viterbi path, we trace the red arcs back from the accept state. In this case the Viterbi path state sequence is $0 \rightarrow 2 \rightarrow 4 \rightarrow 6$ with arc labels b, c, and b.

Example 4.4. Compute the Viterbi path of the graph in figure 4.18.

The Viterbi path is shown in figure 4.19. ■

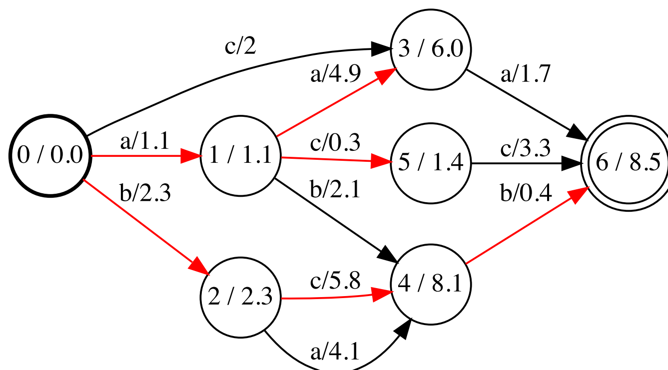


Figure 4.17: Each state is labeled with the state label and Viterbi score from the start state up to that state. The red arrows indicate the arc which is part of the Viterbi path up to the given state. The complete Viterbi path can be found by following red arcs back from the accept state.

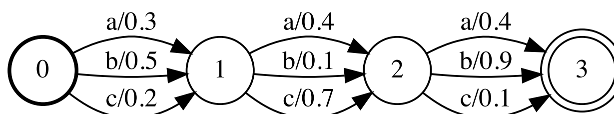


Figure 4.18: An example acceptor for which we would like to compute the Viterbi path.

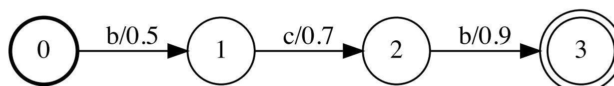


Figure 4.19: The Viterbi path of the graph in figure 4.18.

5 Differentiation with Automata

Modern deep learning relies on the fact that core functions are differentiable (or sub-differentiable). This is important to any type of gradient-based numerical optimization, of which the most commonly used are variations of stochastic gradient descent. The parameters, θ, of the model are specified in tensors. The objective function, $f(\theta, \mathbf{x}, \mathbf{y})$, is a function of the parameters and the input data $\mathbf{x}$ and $\mathbf{y}$. The parameters can then be optimized to improve the objective with variations of a simple update rule:

$$\theta_t = \theta_{t-1} + \alpha \, \nabla_\theta f(\theta_t, \mathbf{x}, \mathbf{y}),$$

where α is the learning rate, and ∇_θ is the gradient operator which we describe in more detail below. This update applies equally well to parameters in graphs as it does to tensors. The main challenge is computing gradients, the subject of this section.

5.1 Derivatives

Many operations used with vectors, matrices, and n-dimensional tensors are differentiable. This means we can compute the change in any of the output elements with respect to an infinitesimal change in any of the input elements. For example, consider a vector $\mathbf{z} = f(\mathbf{x}, \mathbf{y})$ which is the output of a function of two vectors $\mathbf{x}$ and $\mathbf{y}$. The Jacobian of $\mathbf{z}$ with respect to $\mathbf{x}$ is the matrix of partial derivatives with entries $\frac{\partial \mathbf{z}_i}{\partial \mathbf{x}_j}$. The gradient is defined as the tensor of partial derivatives of a scalar function. So if $f(\mathbf{x}) \in \mathbb{R}$ is a scalar function, then the gradient is:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_1}, \ldots, \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_n} \right]^\top.$$

In the same way, we can compute partial derivatives of the arc weights of an output graph for a given operation with respect to the arc weights of any of the input graphs. Take the concatenation operation as an example. Suppose we are given two graphs, $\mathcal{A}$ and $\mathcal{B}$, and we construct the concatenated graph $\mathcal{C} = \mathcal{A}\mathcal{B}$ as in figure 5.1.

For each of the arc weights C_i in the concatenated graph $\mathcal{C}$, we can compute the partial derivative with respect to the arc weights of $\mathcal{A}$ and $\mathcal{B}$. For any arc in $\mathcal{C}$, it

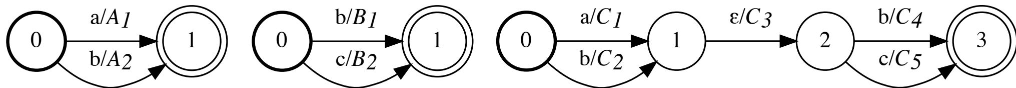


Figure 5.1: The concatenation of the graphs $\mathcal{A}$ (left) and $\mathcal{B}$ (middle) produces $\mathcal{C}$ (right). For each graph the arc weights are shown as variables on the edges.

either has a corresponding arc in $\mathcal{A}$ or $\mathcal{B}$ from which it gets its weight, or it has a weight of zero. The partial derivative of an output arc weight C_i with respect to an input arc weight A_j or B_j is 1 if the two arcs correspond and 0 otherwise. For example, in the graphs in figure 5.1 we have:

$$\frac{\partial C_1}{\partial A_1} = 1, \quad \frac{\partial C_2}{\partial A_2} = 1, \quad \frac{\partial C_4}{\partial B_1} = 1, \quad \text{and} \quad \frac{\partial C_5}{\partial B_2} = 1.$$

The remaining partial derivatives are all 0. For example, for C_1 we have:

$$\frac{\partial C_1}{\partial A_2} = 0, \quad \frac{\partial C_1}{\partial B_1} = 0, \quad \text{and} \quad \frac{\partial C_1}{\partial B_2} = 0.$$

In the following, I use the notation $\frac{\partial \mathcal{C}}{\partial \mathcal{A}}$ to generalize the Jacobian to graphs. This Jacobian is a data structure which contains the partial derivatives $\frac{\partial C_i}{\partial A_j}$ for all arc weights C_i in $\mathcal{C}$ and A_j in $\mathcal{A}$. Another way to view this Jacobian is as a set of graphs $\frac{\partial C_i}{\partial \mathcal{A}}$ indexed by i which are the same size as $\mathcal{A}$. Alternatively we can view the Jacobian as a set of graphs $\frac{\partial \mathcal{C}}{\partial A_j}$ indexed by j which are the same size as $\mathcal{C}$. This is analogous to viewing the Jacobian of a vector-valued function either as a set of columns or a set of rows.

Example 5.1. Compute the partial derivatives of the arc weights of the closure of the graph $\mathcal{A}$ from figure 5.1 with respect to the input arc weights. The closure, $\mathcal{A}^*$, is in figure 5.2.

The non-zero partial derivatives are:

$$\frac{\partial w_2}{\partial A_1} = 1 \quad \text{and} \quad \frac{\partial w_3}{\partial A_2} = 1.$$

The remaining partial derivatives are zero:

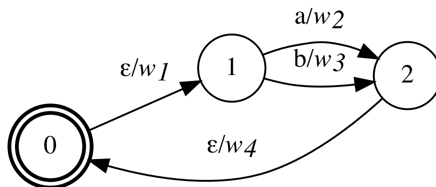


Figure 5.2: The closure of the graph $\mathcal{A}$ from figure 5.1. The weights are denoted by the variables w_i.

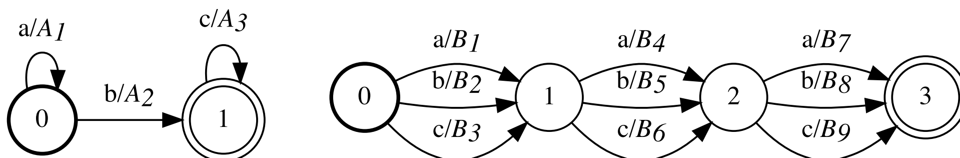


Figure 5.3: We would like to compute the derivative of the intersected graph's arc weights with respect to the arc weights in the two input acceptors $\mathcal{A}$ and $\mathcal{B}$. The arc weights are labeled with the variable names A_j and B_k.

$$\frac{\partial w_1}{\partial A_1} = 0, \quad \frac{\partial w_1}{\partial A_2} = 0, \quad \frac{\partial w_2}{\partial A_2} = 0, \quad \frac{\partial w_3}{\partial A_1} = 0, \quad \frac{\partial w_4}{\partial A_1} = 0, \quad \text{and} \quad \frac{\partial w_4}{\partial A_2} = 0.$$

Example 5.2. Compute the partial derivatives of the intersected automata weights w_i with respect to the input arc weights A_j for graph $\mathcal{A}$ and B_k for graph $\mathcal{B}$ shown in figure 5.3.

The partial derivative $\frac{\partial w_i}{\partial A_j}$ is 1 if the weight w_i came from A_j and zero otherwise. The derivatives with a value of 1 for graph $\mathcal{A}$ are:

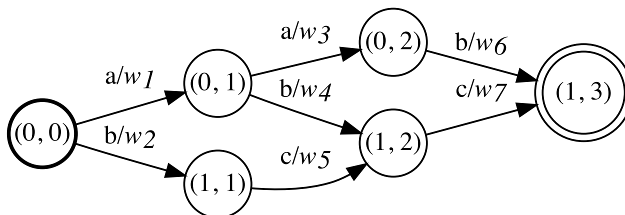


Figure 5.4: The intersected graph of the two acceptors $\mathcal{A}$ and $\mathcal{B}$ in figure 5.3. The weights are denoted as variables w_i on the edges.

$$\frac{\partial w_1}{\partial A_1}, \frac{\partial w_2}{\partial A_2}, \frac{\partial w_3}{\partial A_1}, \frac{\partial w_4}{\partial A_2}, \frac{\partial w_5}{\partial A_3}, \frac{\partial w_6}{\partial A_2}, \text{ and } \frac{\partial w_7}{\partial A_3}.$$

The derivatives with a value of 1 for graph $\mathcal{B}$ are:

$$\frac{\partial w_1}{\partial B_1}, \frac{\partial w_2}{\partial B_2}, \frac{\partial w_3}{\partial B_4}, \frac{\partial w_4}{\partial B_5}, \frac{\partial w_5}{\partial B_6}, \frac{\partial w_6}{\partial B_8}, \text{ and } \frac{\partial w_7}{\partial B_9}.$$

The remaining derivatives for both graphs are zero. ■

5.2 Automatic Differentiation

In the previous section we saw how to compute derivatives for some common automata operations. Automatic differentiation greatly simplifies the process of computing derivatives for arbitrary compositions of operations. In this section, we will discuss *reverse-mode* automatic differentiation at a high-level.

Reverse-mode automatic differentiation proceeds in two steps. First, a forward pass computes all of the operations. Then, a backward pass computes the gradients. During the forward pass the composition of operations is stored in a computation graph (not to be confused with an automata). Data and metadata are also cached during the forward pass to make the gradient computation more efficient. There is often a trade-off between memory and compute in that we can save more intermediate data to reduce the computation required during but increase the memory required during the backward pass.

Consider the graph equation:

$$\text{LSE}\left[((\mathcal{A}_1 + \mathcal{A}_2) \circ \mathcal{X}^*)\right]. \tag{4}$$

This equation can be represented in the computation graph in figure 5.5.

The leaves of the computation graph (the solid circular nodes with no incoming arrows) are either parameter graphs or input data graphs. In this case, let's assume $\mathcal{A}_1$ and $\mathcal{A}_2$ are the parameter graphs, and $\mathcal{X}$ is the graph of input data. The square nodes are operations, and they are always followed by output graphs, which are dashed circular nodes.

During the backward pass, the gradients are computed from the output ($\mathcal{G}_4$ in figure 5.5) following the arrows in the computation graph backwards. A graph can only compute its gradient once all of the graphs downstream of it have had

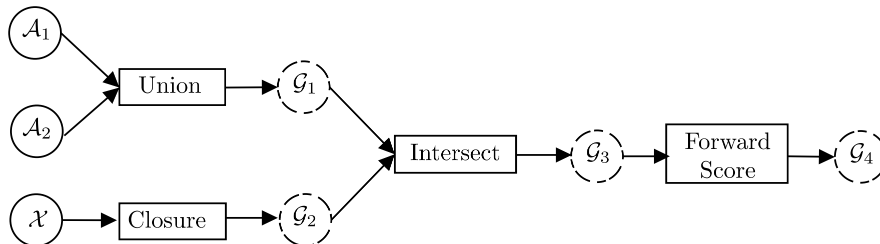


Figure 5.5: An example of a compute graph for equation 4. The solid circular nodes are leaves. The graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ are parameters, and $\mathcal{X}$ is input data. The rectangular nodes are operations. The dashed circular nodes are graphs computed as the result of an operation.

their gradients computed. Thus the backward pass must be done as a reverse topological traversal of the computation graph.

For example, assume the gradient of the output $\mathcal{G}_4$ with respect to graph $\mathcal{G}_1$ has been computed. This gradient $\frac{\partial \mathcal{G}_4}{\partial \mathcal{G}_1}$ is then used to compute $\frac{\partial \mathcal{G}_4}{\partial \mathcal{A}_1}$ and $\frac{\partial \mathcal{G}_4}{\partial \mathcal{A}_2}$. Assuming we know how to differentiate $\mathcal{G}_1$ with respect to $\mathcal{A}_1$ and $\mathcal{A}_2$, then we can compute the desired gradients essentially using the chain rule. Assume g_i are the arc weights of $\mathcal{G}_1$ and a_j are the arc weights of $\mathcal{A}_1$, then the chain rule gives:

$$\frac{\partial \mathcal{G}_4}{\partial a_j} = \sum_i \frac{\partial \mathcal{G}_4}{\partial g_i} \frac{\partial g_i}{\partial a_j}.$$

This is done recursively at every node in the computation graph until the gradients for all the leaf graphs are available.

At a high-level any implementation of reverse-mode automatic differentiation is the same. However, implementations often differ in the details. One simple approach is to have every output graph record the input graphs from which it was generated as well as a gradient computation function. The input graphs and gradient computation function (and any other metadata) can be set during the forward pass by the operation itself.

For example, after the execution of union, the data structure which holds the output graph $\mathcal{G}_1$ could also hold pointers to the inputs $\mathcal{A}_1$ and $\mathcal{A}_2$ and a pointer to the gradient computation function for union. A simplified example of what this data structure might look like is shown in figure 5.6.

Once the gradient for $\mathcal{G}_1$ is available, the union's gradient function is called with

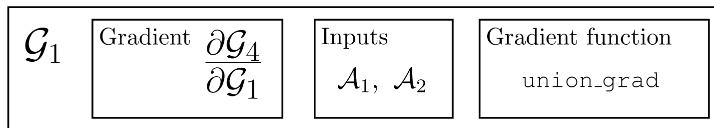


Figure 5.6: The data structure which holds the graph $\mathcal{G}_1$ as well as the data needed to compute the gradient of its inputs. In this case $\mathcal{G}_1$ was the output of a union of $\mathcal{A}_1$ and $\mathcal{A}_2$.

the inputs $\mathcal{A}_1$, $\mathcal{A}_2$, and $\frac{\partial \mathcal{G}_4}{\partial \mathcal{G}_1}$. For $\mathcal{A}_1$, the union gradient function will compute $\frac{\partial \mathcal{G}_1}{\partial \mathcal{A}_1}$ and use the chain rule as described above to assemble the desired gradient $\frac{\partial \mathcal{G}_4}{\partial \mathcal{A}_1}$. It will do the same for $\mathcal{A}_2$.

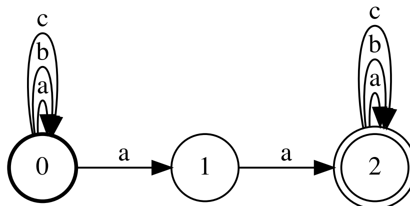


Figure 6.1: A graph which matches the bigram aa.

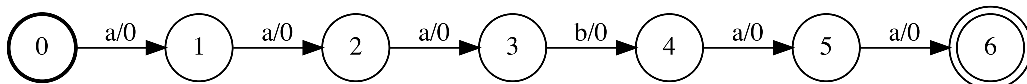


Figure 6.2: A representation of the sequence $aaabaa$ as a linear graph for use in computing the frequency of a given n-gram.

6 Extended Examples

6.1 Counting n-grams

In this example, we'll use graph operations to count the number of n-grams in a string.

Suppose we have a string $aaabaa$ and we want to know the frequency of each bigram. In this case the bigrams contained in the string are aa, ab, and ba with frequencies of 3, 1, and 1 respectively. In the general case we are given an input string and an n-gram and the goal is to count the number of occurrences of the n-gram in the input string.

For a given n-gram, the first step is to construct the graph which matches that n-gram at any location in the string. If $\mathbf{y}$ denotes the n-gram, we want to construct the graph equivalent of the regular expression $. * \mathbf{y} .*$ where $.*$ indicates zero of more occurrences of any token in the token set.

Suppose we want to count the number of occurrences of the bigram aa in $aaabaa$. For the bigram aa, and the token set $\{a, b, c\}$, the n-gram matching graph is shown in figure 6.1. The first and last state allow self-loops on any token in the token set. These states correspond to the $.*$ at the beginning and end of the expression $. * \mathbf{y} .*$.

We can encode the string $aaabaa$ with a simple linear graph, as in figure 6.2.

We then compute the intersection of the graph representing the string and the

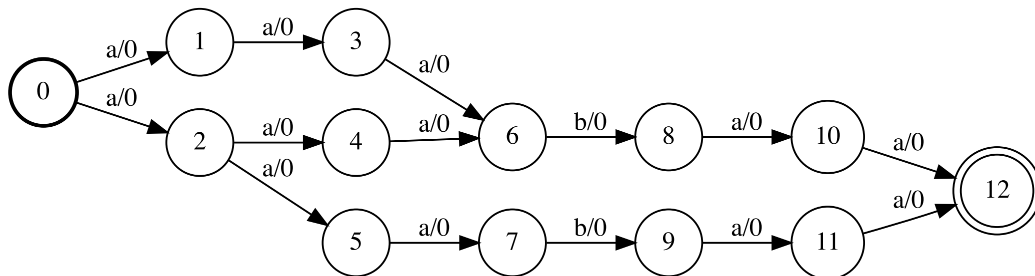


Figure 6.3: The graph represents the intersection of the n-gram graph for aa with the graph representing $aaabaa$. The number of unique paths in this graph, in this case 3, is the frequency of the n-gram ab.

graph representing the bigram. The intersected graph is in figure 6.3. The number of paths in this graph represents the number of occurrences of the bigram in the string.

Since each path has a weight of 0, we can count the number of unique paths in the intersected graph by using the forward score. Assume the intersected graph has p paths. The forward score of the graph is $s = \log \sum_{i=1}^{p} e^0 = \log p$. So the total number of paths is $p = e^s$.

6.2 Edit Distance

In this example we'll use transducers to compute the Levenshtein edit distance between two sequences. The edit distance is a way to measure the similarity between two sequences by computing the minimum number of operations required to change one sequence into the other. The Levenshtein edit distance allows for insertion, deletion, and substitution operations.

For example, consider the two strings "saturday" and "sunday". The edit distance between them is 3. One way to minimally edit "saturday" to "sunday" is with two deletions (`D`) and a substitution (`S`) as below:

```
s a t u r d a y
  D D   S
s     u n d a y
```

We can compute the edit distance between two strings with the use of transducers. The idea is to transduce the first string into the second according to the allowed operations encoded as a graph.

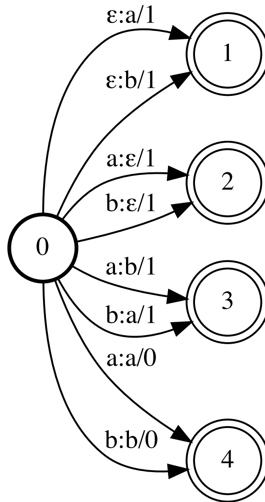


Figure 6.4: An example of an edits graph $\mathcal{E}$ for the token set $\{a, b\}$. The graph encodes the allowed edit distance operations and their associated penalties.

We first construct an edits graph $\mathcal{E}$ which encodes the allowed operations. An example of an edits graph assuming a token set of $\{a, b\}$ is shown in figure 6.4. The insertion of a token is represented by the arcs from state 0 to state 1 and has a cost of 1. The deletion of a token is represented by the arcs from state 0 to state 2 which also incur a cost of 1. All possible substitutions are encoded in the arcs from 0 to 3 and again have a cost of 1. We also have to encode the possibility of leaving a token unchanged. This is represented on the arcs from 0 to 4, and the cost is 0.

We then take closure of the edits graph $\mathcal{E}$ to represent the fact that we can make zero or more of any of the allowed edits. We then encode the first sequence $\mathbf{x}$ in a graph $\mathcal{X}$ and the second sequence $\mathbf{y}$ in a graph $\mathcal{Y}$. All possible ways of editing $\mathbf{x}$ to $\mathbf{y}$ can be computed by taking the composition:

$$\mathcal{P} = \mathcal{X} \circ \mathcal{E}^* \circ \mathcal{Y}.$$

The graph $\mathcal{P}$ represents the set of all possible unique ways we can edit the sequence $\mathbf{x}$ into $\mathbf{y}$. The score of a given path in $\mathcal{P}$ is the associated cost. We can then find the edit distance by computing the path with the smallest score in $\mathcal{P}$. For this, we could use the Viterbi algorithm with a min instead of a max. Alternatively, we can use weights of -1 instead of 1 in $\mathcal{E}$ and use the Viterbi algorithm unchanged. In this case, the path with the largest score (the one with the least negative score)

represents the edit distance. The actual edits (*i.e.* the insertions, deletions, and substitutions) can be found by computing the Viterbi path.

Example 6.1 (). Compute the edit distance graph $\mathcal{P}$ between $\mathbf{x} = aba$ and $\mathbf{y} = aabb$, the edit distance between the two sequences, and one possible set of operations which attain the edit distance.

Proof. Using an equivalent but more compact representation of the edit distance graph $\mathcal{E}^*$ yields the graph $\mathcal{P} = \mathcal{X} \circ \mathcal{E}^* \circ \mathcal{Y}$, shown in figure 6.5. Each path in $\mathcal{P}$ represents a unique conversion of $\mathcal{X}$ into $\mathcal{Y}$ using insertion, deletion, and substitution operations. The negation of the score of the path is the number of such operations required.

For example, the path along the state sequence $0 \rightarrow 1 \rightarrow 6 \rightarrow 11 \rightarrow 16$ converts $\mathbf{x}$ to $\mathbf{y}$ with a distance of two using an insertion at the second letter and a substitution at the end:

```
a     b  a
   I     S
a  a  b  b
```

The Viterbi score and Viterbi path yield the edit distance between $\mathbf{x}$ and $\mathbf{y}$ and the sequence of edit operations required to attain the edit distance. The Viterbi path for the example is shown in figure 6.6.

■

6.3 n-gram Language Model

In this example we will encode an n-gram language model as an acceptor. We will then use the acceptor to compute the language model probability for a given sequence.

Let's start with a very simple example. Suppose we have the token set $\{a, b, c\}$ and we want to construct a unigram language model. Given counts of occurrences for each token in the vocabulary, we can construct an acceptor to represent the unigram language model. Suppose we are given the probabilities 0.5, 0.2, and 0.3 for a, b, and c respectively. The corresponding unigram graph is shown in figure 6.7. Note that the edge weights are log probabilities.

Now assume we are given the sequence aa for which we would like to compute the probability. The probability under the language model is $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$. We can

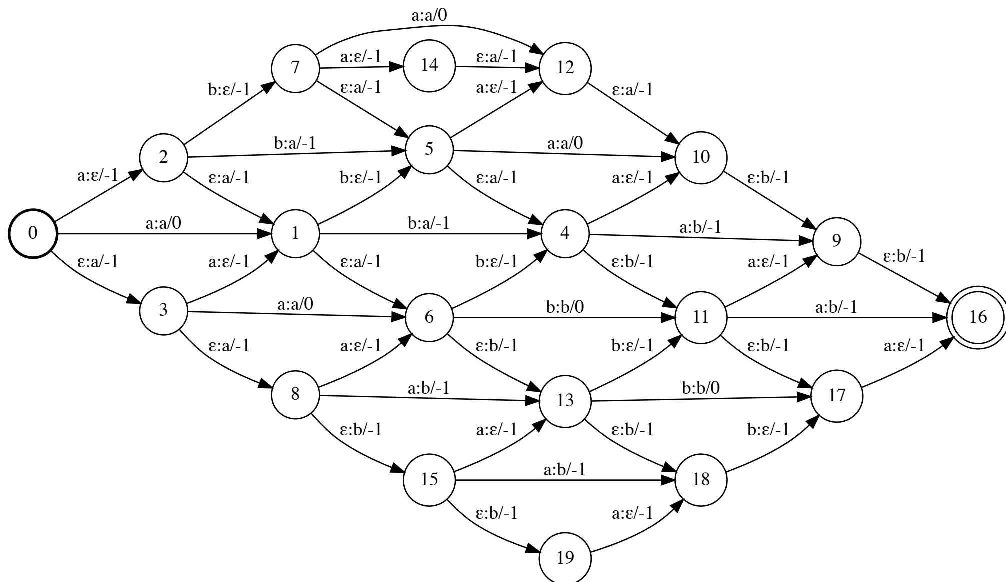


Figure 6.5: The graph $\mathcal{P} = \mathcal{X} \circ \mathcal{E}^* \circ \mathcal{Y}$ represents all possible ways to convert $\mathbf{x} = aba$ to $\mathbf{y} = aabb$ using insertions, deletions, and substitutions. Since the penalties are negative, the score of the highest scoring path is the edit distance between the two sequences. The path itself encodes the sequence of operations required.

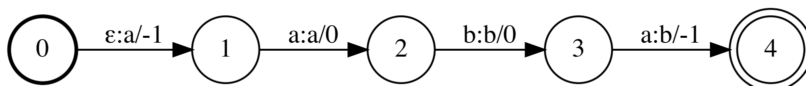


Figure 6.6: The Viterbi path of the graph $\mathcal{P} = \mathcal{X} \circ \mathcal{E}^* \circ \mathcal{Y}$. The edit distance is 2 with one insertions (the arc between states 0 and 1) and one substitution (the arc between states 3 and 4).

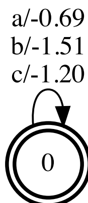


Figure 6.7: A unigram graph $\mathcal{U}$ for $\{a, b, c\}$.

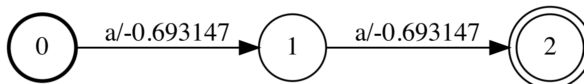


Figure 6.8: The sequence aa scored with the unigram log probabilities from the graph $\mathcal{U}$ in figure 6.7.

compute the log probability of aa by intersecting its graph representation $\mathcal{X}$ with the unigram graph $\mathcal{U}$ and then computing the forward score:

$$\log p(aa) = \text{LSE}(\mathcal{X} \circ \mathcal{U})$$

The graph in figure 6.8 shows the intersection $\mathcal{X} \circ \mathcal{U}$. The arc edges in the intersected graph contain the correct unigram scores, and the forward score gives the log probability of the sequence aa. In this case, the Viterbi score would give the same result since the graph has only one path.

For an arbitrary sequence $\mathbf{x}$ with a graph representation $\mathcal{X}$ and an arbitrary n-gram language model with graph representation $\mathcal{N}$, the log probability of $\mathbf{x}$ is given by:

$$\log p(\mathbf{x}) = \text{LSE}(\mathcal{X} \circ \mathcal{N}).$$

Next, let's see how to represent a bigram language model as a graph. From there, the generalization to arbitrary order n is relatively straightforward. Assume again we have the token set $\{a, b, c\}$. The bigram model is shown in the graph in figure 6.9. Each state is labeled with the token representing the most recently seen input. For a bigram model we only need to remember the previous token to know which score to use when processing the next token. For a trigram model we would need to remember the previous two tokens. For an n-gram model we would need to remember the previous $n - 1$ tokens. The label and score pair leaving each state represent the corresponding conditional probability (technically these should be log probabilities). Each state has an outgoing arc for every possible token in the token set.

Example 6.2. Compute the number of states and arcs in a graph representation of an n-gram language model for a given order n and a token set size of v.

For order n, the graph needs a state for every possible token sequence of length $n-1$. This means that the graph will have v^{n-1} states. Each state has v outgoing arcs. Thus the total number of arcs in the graph is $v \cdot v^{n-1} = v^n$. This should be expected given that the language model assigns a score for every possible sequence of length n. ■

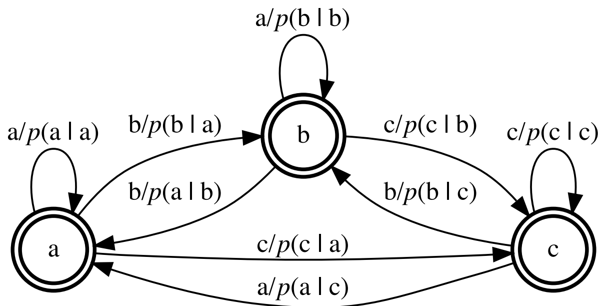


Figure 6.9: A bigram model for the token set $\{a, b, c\}$. Each arc is labeled with the next observed token and the corresponding bigram probability.

6.4 Automatic Segmentation Criterion

In machine-learning applications with sequential data, we often need to compute a conditional probability of an output sequence given an input sequence when the two sequences do not have the same length. The Automatic Segmentation criterion (ASG) is one of several common loss functions for which this is possible. However, ASG is limited to the case when the output sequence is no longer than the input sequence.

Assume we have an input sequence of T vectors $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_T]$ and an output sequence of U tokens, $\mathbf{y} = [y_1, \ldots, y_U]$ such that $U \leq T$. We don't know the actual alignment between $\mathbf{X}$ and $\mathbf{y}$, and in many applications we don't need it. For example, in speech recognition $\mathbf{X}$ consists of frames of Fourier-transformed audio, and $\mathbf{y}$ could be letters of a transcript. We usually don't need to know how $\mathbf{y}$ aligns to $\mathbf{X}$; we only require that $\mathbf{y}$ is the correct transcript. To get around not knowing this alignment, the ASG criterion marginalizes over all possible alignments between $\mathbf{X}$ and $\mathbf{y}$.

In ASG, the output sequence is aligned to a given input by allowing one or more consecutive repeats of each token in the output. Suppose we have an input of length 5 and the output sequence *ab*. Some possible alignments of the output are *aaabb*, *abbbb*, and *aaaab*. Some invalid alignments are *abbba*, *aaab*, and *aaaaa*. These are invalid because the first corresponds to the output *aba*, the second is too short, and the third corresponds to the output *a*.

For each time-step of the input, we have a model $s_t(a)$ which assigns a score for every possible output token a. Note the model $s_t(\cdot)$ is conditioned on some or all of $\mathbf{X}$, but I won't include this in the notation for simplicity. Let $\mathbf{a} = [a_1, \ldots, a_T]$

be one possible alignment between $\mathbf{X}$ and $\mathbf{y}$. The alignment $\mathbf{a}$ also has length T. To compute a score for $\mathbf{a}$, we sum the sequence of scores for each token:

$$s(\mathbf{a}) = \sum_{t=1}^{T} s_t(a_t)$$

Let $\mathcal{A}_{\mathbf{X},\mathbf{y}}$ denote the set of all possible alignments between $\mathbf{X}$ and $\mathbf{y}$. We use the individual alignment scores to compute a conditional probability of the output $\mathbf{y}$ given the input $\mathbf{X}$ by marginalizing over $\mathcal{A}_{\mathbf{X},\mathbf{y}}$:

$$\log p(\mathbf{y} \mid \mathbf{X}) = \underset{\mathbf{a} \in \mathcal{A}_{\mathbf{X},\mathbf{y}}}{\text{LSE}} \; s(\mathbf{a}) - \log Z.$$

The normalization term Z is given by:

$$Z = \sum_{\mathbf{a} \in \mathcal{Z}_{\mathbf{X}}} e^{s(\mathbf{a})},$$

where $\mathcal{Z}_{\mathbf{X}}$ is the set of all possible length T alignments (the same length as X). Computing the summations over $\mathcal{A}_{\mathbf{X},\mathbf{y}}$ and $\mathcal{Z}_{\mathbf{X}}$ explicitly is not tractable because the sizes of these sets grow rapidly with the lengths of $\mathbf{X}$ and $\mathbf{y}$. Let's instead use automata to encode these sets and efficiently compute the summation using the forward score operation. I will use the script variables $\mathcal{A}_{\mathbf{X},\mathbf{y}}$ and $\mathcal{Z}_{\mathbf{X}}$ to represent both sets of sequences and the analogous graph. It will be clear from context which representation is intended.

Let's start with the normalization term Z. The set $\mathcal{Z}_{\mathbf{X}}$ encodes all possible outputs of length T, where T is the length of $\mathbf{X}$. As an example, assume $T = 4$ and we have three possible output tokens $\{a, b, c\}$. If the scores for each output are independent, we can represent $\mathcal{Z}_{\mathbf{X}}$ with the graph in figure 6.10. The scores on the arcs are given by the model $s_t(\cdot)$. These scores are often called the *emissions*, and the graph itself is sometimes called the emissions graph. I'll use $\mathcal{E}$ to represent the emissions graph. In this case the emissions graph $\mathcal{E}$ is the same as the normalization graph $\mathcal{Z}_{\mathbf{X}}$; however, in general they may be different. The log normalization term is the forward score of the emissions graph, $\log Z = \text{LSE}(\mathcal{E})$.

Let's turn to the set $\mathcal{A}_{\mathbf{X},\mathbf{y}}$ which we will also represent as an acceptor. This acceptor should have a path for every possible alignment between $\mathbf{X}$ and $\mathbf{y}$. We'll construct $\mathcal{A}_{\mathbf{X},\mathbf{y}}$ in two steps. First, we can encode the set of allowed alignments of arbitrary length for a given sequence $\mathbf{y}$ with a graph, $\mathcal{A}_{\mathbf{y}}$. As an example, the graph $\mathcal{A}_{\mathbf{y}}$ for the sequence ab is shown in figure 6.11.

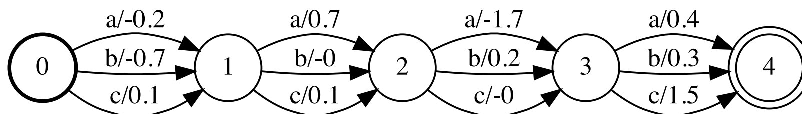


Figure 6.10: An emissions graph $\mathcal{E}$ with $T = 4$ time-steps and a token set of $\{a, b, c\}$.

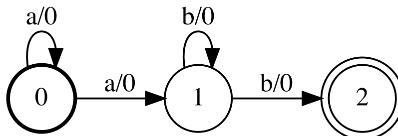


Figure 6.11: The ASG alignment graph $\mathcal{A}_{\mathbf{y}}$ for the sequence ab. The graph encodes the fact that each output token can repeat one or more times in an arbitrary length alignment.

The graph in figure 6.11 has a simple interpretation. Each token in the output ab can repeat one or more times in the alignment. We can then construct $\mathcal{A}_{\mathbf{X},\mathbf{y}}$ by intersecting $\mathcal{A}_{\mathbf{y}}$ with the emissions graph $\mathcal{E}$, which represents all possible sequences of length T. This gives $\mathcal{A}_{\mathbf{X},\mathbf{y}} = \mathcal{A}_{\mathbf{y}} \circ \mathcal{E}$. An example of $\mathcal{A}_{\mathbf{X},\mathbf{y}}$ is shown in figure 6.12 for the sequence ab with $T = 4$.

In terms of graph operations, we can write the ASG criterion as:

$$\log p(\mathbf{y} \mid \mathbf{X}) = \text{LSE}(\mathcal{A}_{\mathbf{y}} \circ \mathcal{E}) - \text{LSE}(\mathcal{E}). \tag{5}$$

Global or Local Normalization

The ASG criterion is *globally normalized*. The term Z is the global normalization term. It ensures that the conditional probability $p(\mathbf{y} \mid \mathbf{X})$ distribution is valid in that it sums to one over $\mathbf{y}$. The global normalization term Z (also known as the partition function) is often the most expensive

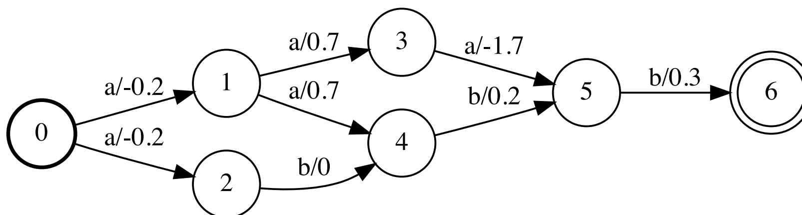


Figure 6.12: The alignment graph $\mathcal{A}_{\mathbf{X},\mathbf{y}}$ for the input $\mathbf{X}$ with $T = 4$ time-steps and an output $\mathbf{y} = ab$.

part of the loss to compute.

In some cases the global normalization can be avoided by using a *local normalization*. For example, in the transition-free version of ASG described above, the path score for **a** decomposes into a separate score for each time-step. In this case, we can compute the exact same loss by normalizing the scores $s_t(a)$ at each time-step and dropping the term Z. Concretely, we compute the normalized scores at each time-step:

$$p_t(a) = \frac{e^{s_t(a)}}{\sum_z e^{s_t(z)}}.$$

We then replace the unnormalized scores with the log-normalized scores when computing the score for an alignment:

$$\log p(\mathbf{a}) = \sum_{t=1}^{T} \log p_t(a_t).$$

As a last step, we remove the global normalization term Z from the loss function, but leave it otherwise unchanged:

$$\log p(\mathbf{y} \mid \mathbf{X}) = \operatorname*{LSE}_{\mathbf{a} \in \mathcal{A}_{\mathbf{X},\mathbf{y}}} \log p(\mathbf{a}).$$

In the version which uses graph operations, the log-normalized scores $\log p_t(y)$ become the arc weights of the emissions graph $\mathcal{E}$. The graph based loss function then simplifies to:

$$\log p(\mathbf{y} \mid \mathbf{X}) = \text{LSE}(\mathcal{A}_{\mathbf{y}} \circ \mathcal{E}).$$

We can prove that the locally normalized version of the transition-free ASG loss is equivalent to the globally normalized version. To do this, we need to show:

$$\operatorname*{LSE}_{\mathbf{a} \in \mathcal{A}_{\mathbf{X},\mathbf{y}}} \log p(\mathbf{a}) = \operatorname*{LSE}_{\mathbf{a} \in \mathcal{A}_{\mathbf{X},\mathbf{y}}} s(\mathbf{a}) - \log Z \tag{6}$$

To show this, we need two identities. The first identity lets us pull independent terms out of the LSE operation:

$$\operatorname*{LSE}_{x}(x + y) = \text{LSE}(x) + y.$$

The proof of this identity is below:

$$\operatorname{LSE}_x(x+y) = \log \sum_x e^{x+y} = \log \sum_x e^x e^y = \log \sum_x e^x + \log e^y = \operatorname{LSE}(x) + y.$$

The second identity lets us rearrange products and sums:

$$\prod_{t=1}^{T} \sum_z s_t(z) = \sum_{\mathbf{z}} \prod_{t=1}^{t} s_t(z_t).$$

The term on the left is the product over t of the sum of $s_t(z)$ over all possible values of the token z. The term on the right is sum over all possible token sequences $\mathbf{z}$ of length T of the product scores $s_t(z_t)$ for each time-step in the sequence. A short proof is below:

$$\begin{aligned}
\prod_{t=1}^{T} \sum_z s_t(z) &= \left(\sum_z s_1(z)\right) \ldots \left(\sum_z s_T(z)\right) \\
&= \sum_{z_1} \ldots \sum_{z_T} \prod_{t=1}^{T} s_t(z_t) \\
&= \sum_{\mathbf{z}} \prod_{t=1}^{T} s_t(z_t).
\end{aligned}$$

We are now ready to verify equation 6. Starting from the left hand side of equation 6, we have:

$$\begin{aligned}
\operatorname*{LSE}_{\mathbf{a} \in \mathcal{A}_{\mathbf{X},\mathbf{y}}} \log p(\mathbf{a}) &= \operatorname*{LSE}_{\mathbf{a} \in \mathcal{A}_{\mathbf{X},\mathbf{y}}} \sum_{t=1}^{T} \log p_t(a_t) \\
&= \operatorname*{LSE}_{\mathbf{a} \in \mathcal{A}_{\mathbf{X},\mathbf{y}}} \sum_{t=1}^{T} \log \frac{e^{s_t(a_t)}}{\sum_z e^{s_t(z)}} \\
&= \operatorname*{LSE}_{\mathbf{a} \in \mathcal{A}_{\mathbf{X},\mathbf{y}}} \left(\sum_{t=1}^{T} s_t(a_t) - \sum_{t=1}^{T} \log \sum_z e^{s_t(z)} \right).
\end{aligned}$$

Using the first identity, we get:

$$\operatorname*{LSE}_{\mathbf{a}\in\mathcal{A}_{\mathbf{X},\mathbf{y}}} \log p(\mathbf{a}) = \operatorname*{LSE}_{\mathbf{a}\in\mathcal{A}_{\mathbf{X},\mathbf{y}}} \left(\sum_{t=1}^{T} s_t(a_t)\right) - \sum_{t=1}^{T} \log \sum_{z} e^{s_t(z)}$$

The first term on the right is:

$$\operatorname*{LSE}_{\mathbf{a}\in\mathcal{A}_{\mathbf{X},\mathbf{y}}} \left(\sum_{t=1}^{T} s_t(a_t)\right) = \operatorname*{LSE}_{\mathbf{a}\in\mathcal{A}_{\mathbf{X},\mathbf{y}}} s(\mathbf{a}).$$

Using the second identity, the second term on the right becomes:

$$\begin{aligned}
\sum_{t=1}^{T} \log \sum_{z} e^{s_t(z)} &= \log \prod_{t=1}^{T} \sum_{z} e^{s_t(z)} \\
&= \log \sum_{\mathbf{z}\in\mathcal{Z}_{\mathbf{X}}} \prod_{t=1}^{T} e^{s_t(z_t)} \\
&= \log \sum_{\mathbf{z}\in\mathcal{Z}_{\mathbf{X}}} e^{\sum_{t=1}^{T} s_t(z_t)} \\
&= \log \sum_{\mathbf{z}\in\mathcal{Z}_{\mathbf{X}}} e^{s(\mathbf{z})} \\
&= \log Z.
\end{aligned}$$

Putting these two terms together yields the right hand side of equation 6.

6.4.1 Transitions

The original ASG loss function also includes bigram transition scores. The alignment score with transitions, $h(a_{t-1}, a_t)$, included is given by:

$$s(\mathbf{a}) = \sum_{t=1}^{T} s_t(a_t) + h(a_t, a_{t-1}),$$

where a_0 is a special start of sequence token ¡s¿. We can use the alignment scores in the same manner as above and the rest of the loss function is unchanged.

Let's see how to incorporate transitions using an acceptor and graph operations.

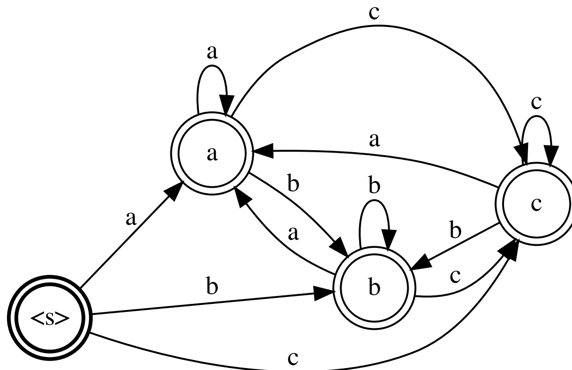


Figure 6.13: The acceptor represents a bigram transition model for the token set $\{a, b, c\}$ (the arc weights are not shown). Each path begins in the start state denoted by the start-of-sequence symbol <s>.

I'll rely on the ideas introduced in section 6.3 on n-gram language models, so now is a good time to review that section. The first step is to encode the bigram model as a graph, as in figure 6.13.

To incorporate transition scores for the output sequence, we intersect the bigram graph $\mathcal{B}$ with the output alignment graph $\mathcal{A}_{\mathbf{y}}$. To incorporate transition scores in the normalization term Z, we intersect $\mathcal{B}$ with the emissions graph $\mathcal{E}$. The loss function using graph operations including transitions becomes:

$$\log p(\mathbf{y} \mid \mathbf{X}) = \text{LSE}(\mathcal{B} \circ \mathcal{A}_{\mathbf{y}} \circ \mathcal{E}) - \text{LSE}(\mathcal{B} \circ \mathcal{E}).$$

We see here an example of the expressive power of a graph-based implementation of the loss function. In a non-graph-based implementation of ASG, the use of a bigram transition function is hard-coded. In the graph-based version, the transition model $\mathcal{B}$ could be a unigram, bigram, trigram, or otherwise arbitrary n-gram. Of course, the size of the transition graph increases rapidly with the order n and the size of the token set (see example 6.2). This causes problems with both sample and computational efficiency. Thus, in practice ASG is used with a bigram transition model and token set sizes rarely larger than a hundred.

The arc weights on the transition graph (the scores $h(a_t, a_{t-1})$), are typically parameters of the model and learned directly. This means we need to compute the gradient of the ASG loss with respect to these scores. These derivatives are straightforward to compute in a framework with automatic differentiation.

```python
def ASG(E, B, Ay):
  # Compute constrained and normalization graphs:
  AXy = gtn.intersect(gtn.intersect(B, Ay), E)
  ZX = gtn.intersect(B, E)

  # Forward both graphs:
  AXy_score = gtn.forward_score(AXy)
  ZX_score = gtn.forward_score(ZX)

  # Compute the loss:
  loss = gtn.negate(gtn.subtract(AXy_score, ZX_score))

  # Clear the previous gradients:
  E.zero_grad()
  B.zero_grad()

  # Compute gradients:
  gtn.backward(loss, retain_graph=False)

  # Return the loss and the gradients:
  return loss.item(), E.grad(), B.grad()
```

Figure 6.14: An example implementation of the ASG loss function which takes as input the emissions graph $\mathcal{E}$, the transitions graph $\mathcal{B}$, and the target alignment graph $\mathcal{A}_{\mathbf{y}}$. The implementation uses the GTN framework.

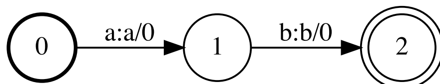


Figure 6.15: The graph $\mathcal{Y}$ corresponding to the target sequence $\mathbf{y} = ab$.

An example implementation of the ASG loss function in the GTN framework [3] is shown in figure 6.14. This function takes as input an emissions graph $\mathcal{E}$, a transitions graph $\mathcal{B}$, and an output alignment graph $\mathcal{A}_\mathbf{y}$ is shown below. I would like to make three observations about this code:

1. The implementation is concise. Given the appropriate graph inputs, the complete loss function requires only eight short lines using ten function calls.
2. The code should look familiar to users of tensor-based frameworks like PyTorch. Other than the different operation names, the imperative style and gradient computation is no different with graphs than it is with tensors.
3. The code is generic. For example, we can pass a trigram model as the input graph argument 'B' without needing to change the function.

6.4.2 ASG with Transducers

As a final step, I'll show how to construct the ASG criterion from even simpler transducer building blocks. The advantage of this approach is that it lets us easily experiment with changes to the criterion at a deeper level.

Our goal is to compute $\mathcal{A}_\mathbf{y}$ from simpler graphs instead of hard-coding its structure directly. The simpler graphs will represent the individual tokens and the target sequence $\mathbf{y}$.

The target sequence graph $\mathcal{Y}$ is a simple linear-chain graph with arc labels taken consecutively from the sequence $\mathbf{y}$. The graph in figure 6.15 shows an example for the sequence ab.

Next we construct the individual token graphs. These graphs encode the assumption that each token in an output maps to one or more repeats of the same token in an alignment. For example for the output $\mathbf{y} = ab$ and alignment $\mathbf{a} = aaaabb$ the token a maps to $aaaa$ and b maps to bb. For the token a, the token graph $\mathcal{T}_a$ shown in figure 6.16 has the desired property.

[3] The GTN framework is open source and available at `https://github.com/gtn-org/gtn`

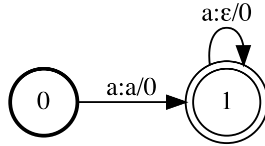


Figure 6.16: An individual token graph $\mathcal{T}_a$ for the token a. The graph encodes the fact that the output a can map to one or more repeats in the alignment.

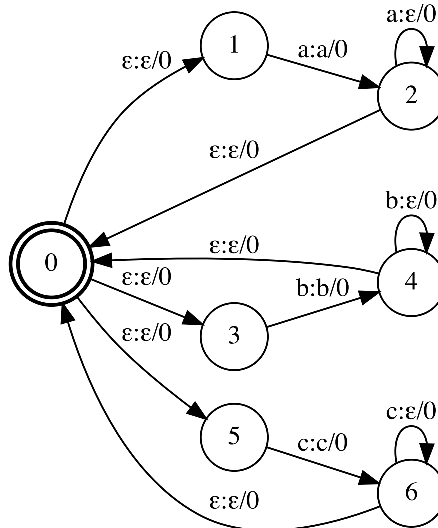


Figure 6.17: The complete token graph $\mathcal{T}$ for the token set $\{a, b, c\}$. The token graph is constructed by taking the closure of the union of the individual token graphs, $\mathcal{T} = (\mathcal{T}_a + \mathcal{T}_b + \mathcal{T}_c)^*$.

Since an output sequence can consist of any sequence of tokens from the token set, we construct the complete token graph $\mathcal{T}$ by taking the union of the individual token graphs and computing the closure. If we have a token set $\{a, b, c\}$, then we construct individual token graphs $\mathcal{T}_a$, $\mathcal{T}_b$, and $\mathcal{T}_c$. The complete token graph $\mathcal{T}$ is given by $\mathcal{T} = (\mathcal{T}_a + \mathcal{T}_b + \mathcal{T}_c)^*$. The complete token graph is shown in figure 6.17.

The graph $\mathcal{A}_{\mathbf{y}}$ can then be constructed by composing $\mathcal{T}$ and $\mathcal{Y}$. In other words, $\mathcal{A}_{\mathbf{y}} = \mathcal{T} \circ \mathcal{Y}$. We have to be careful here. The graphs $\mathcal{T}$ and $\mathcal{Y}$ are transducers and their order in the composition makes a difference. Because of the way we constructed them, we will only get the correct $\mathcal{A}_{\mathbf{y}}$ if $\mathcal{T}$ is the first argument to the composition.

At this point, we can compute the ASG loss just as before using equation 5. The remaining graphs $\mathcal{E}$ and $\mathcal{B}$ are unchanged.

This decomposition of the ASG loss using simple graph building blocks makes it easier to change. In the following section, I will show how to construct a different algorithm, Connectionist Temporal Classification, with only a minor modification to the ASG loss.

6.5 Connectionist Temporal Classification

Connectionist Temporal Classification (CTC) is another loss function which assigns a conditional probability $p(\mathbf{y} \mid \mathbf{X})$ when the input length T and output length U are variable, and the alignment between them is unknown. Like ASG, CTC gets around the fact that the alignment is unknown by assuming a set of allowed alignments and marginalizing over this set. In CTC, the length of the output $\mathbf{y}$ is also constrained in terms of the length of the input. For CTC, the output length U must satisfy $U + R_{\mathbf{y}} \leq T$, where $R_{\mathbf{y}}$ is the number of consecutive repeats in $\mathbf{y}$.

The ASG loss function has two modeling limitations which CTC attempts to improve. First, ASG does not elegantly handle repeat tokens in the output. Second, ASG does not allow for optional null inputs. I'll discuss each of these in turn.

Repeat Tokens: A repeat token is two consecutive tokens with the same value. For example, the b's in $abba$ is a repeat, but the a is not. In ASG, repeat tokens in the output create an ambiguity. A single alignment can map to multiple output sequences. For example, consider the alignment $abbbaa$. This can map to any of the outputs aba, $abba$, $abbba$, $abaa$, $abbaa$, or $abbbaa$. There are several heuristics to resolve this. One option is to use special tokens for repeat characters. For example if $\mathbf{y} = abba$, then we could encode it as ab_2a, where b_2 corresponds to two b's. In this case, the alignment $abbba$ corresponds to the output aba, the alignment $ab_2b_2b_2a$ corresponds to the output $abba$, and the alignment $abbb_2a$ corresponds to $abbba$.

There are two problems with this solution. First, we have to hard code into the token set an upper limit on the number of repeats to allow. Second, if we allow n repeats, then we multiply the token set size by a factor of n causing potential computation and sample efficiency issues.

Blank Inputs: The second problem with ASG is it dos not allow optional null inputs. Any output token y_u must map to a corresponding input $\mathbf{x}_t$. In some cases, the y_u may not meaningfully correspond to any input. The blank token in

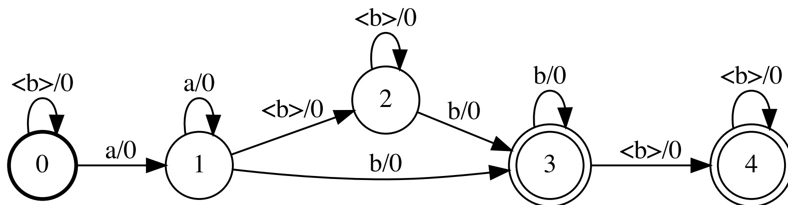


Figure 6.18: The CTC alignment graph $\mathcal{A}_{\mathbf{y}}$ for the sequence *ab*. The graph encodes the fact that each output token can repeat one or more with an optional <b> at the beginning, end, or in between *a* and *b*.

CTC allows for this by representing an input time step which does not correspond to any output.

I'll denote the CTC blank token with <b>. The token <b> can appear zero or more times in the alignment, and it can be at the beginning, in between, or end of the tokens of the output $\mathbf{y}$. If $\mathbf{y}$ has consecutive repeats, then there must be at least one <b> between them in any alignment. So the optional blank token is a solution to both the modeling of null input time steps as well as repeat tokens in the output. Note also that non-optional blank between repeat tokens is why in CTC the output length must satisfy $U + R_{\mathbf{y}} < T$.

As an example, suppose we have an input of length 5 and the output sequence *abb*. Some allowed alignments are *abb*<b>*b*, *ab*<b>*bb*, or *aab*<b>*b*. Some alignments which are not allowed are *aabbb* and *aa*<b>*bb*, both of which correspond to the output *ab* instead of *abb*.

In equations, CTC is indistinguishable from ASG without transitions. The loss is given by:

$$\log p(\mathbf{y} \mid \mathbf{X}) = \operatorname*{LSE}_{\mathbf{a} \in \mathcal{A}_{\mathbf{X},\mathbf{y}}} \log p(\mathbf{a}).$$

The distinction between ASG and CTC is in the set of allowed alignments $\mathcal{A}_{\mathbf{X},\mathbf{y}}$. Assuming we have log normalized scores on the arc weights of the emissions graph $\mathcal{E}$, the graph-based CTC loss is:

$$\log p(\mathbf{y} \mid \mathbf{X}) = \text{LSE}(\mathcal{A}_{\mathbf{y}} \circ \mathcal{E}),$$

where the distinction from ASG is in the graph $\mathcal{A}_{\mathbf{y}}$. An example alignment graph $\mathcal{A}_{\mathbf{y}}$ for CTC for the sequence *ab* is shown in figure 6.18.

The CTC alignment graph encodes the assumptions regarding the blank token, <b>. The alignment can optionally start with <b> as in state 0. The alignment

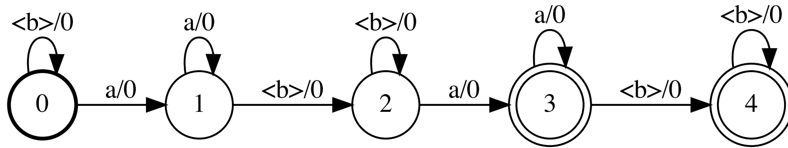


Figure 6.19: The CTC alignment graph $\mathcal{A}_{\mathbf{y}}$ for the sequence aa.

can optionally end in <b> since both state 3 and 4 are accept states. And lastly, the blank is optional between a and b since there is both an arc between states 1 and 3 and a path through state 2 which emits a <b>.

Example 6.3. Construct the CTC $\mathcal{A}_{\mathbf{y}}$ graph for the sequence aa.

The graph $\mathcal{A}_{\mathbf{y}}$ for the sequence $\mathbf{y} = aa$ is shown in figure 6.19. Notice the <b> token in between the first and second a is not optional. The only difference between the graph for aa and the graph for ab is the removal of the arc between states 1 and 3. ■

6.5.1 CTC from Transducers

Like ASG we can construct the graph $\mathcal{A}_{\mathbf{y}}$ used in CTC from smaller building blocks. In fact, one of the motivations of decomposing ASG into simpler transducer building blocks is that it makes constructing CTC almost trivial. To get CTC, we just need to add the <b> token to the tokens graph with the correct semantics. The <b> token graph is a single start and accept state which encodes the fact that the blank is optional. The state has a self-loop which transduces <b> to ϵ, since <b> never yields an output token.

Recall for ASG with the alphabet $\{a, b, c\}$, the graph $\mathcal{A}_{\mathbf{y}}$ is given by:

$$\mathcal{A}_{\mathbf{y}} = (\mathcal{T}_a + \mathcal{T}_b + \mathcal{T}_c)^* \circ \mathcal{Y}.$$

Assuming $\mathcal{T}_{\text{<b>}}$ represents the <b> token transducer as described above, the CTC graph $\mathcal{A}_{\mathbf{y}}$ is given by:

$$\mathcal{A}_{\mathbf{y}} = (\mathcal{T}_a + \mathcal{T}_b + \mathcal{T}_c + \mathcal{T}_{\text{<b>}})^* \circ \mathcal{Y}.$$

The equation above shows how CTC is really the result of one core additional building block encoding the correct behavior of the <b> token. There is one caveat which is that the equation does not force the <b> token in between repeats, so they are not handled correctly. Encoding this constraint through operations on the simpler transducers requires more work but is certainly doable.

Acknowledgements

Thanks to Wes Bouaziz, Fangjun Kuang, and Loren Lugosch for feedback on this work.

References

[1] k2. https://github.com/k2-fsa/k2, 2020.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[3] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. Openfst: A general and efficient weighted finite-state transducer library. In *International Conference on Implementation and Application of Automata*, pages 11–23. Springer, 2007.

[4] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, pages 1–7. Austin, TX, 2010.

[5] Léon Bottou, Yann Le Cun, and Yoshua Bengio. Global training of document processing systems using graph transformer networks. In *Proceedings of Computer Vision and Pattern Recognition (CVPR)*, pages 489–493, Puerto-Rico, 1997. IEEE. URL http://leon.bottou.org/papers/bottou-97.

[6] Thomas M Breuel. The OCRopus open source OCR system. In *Document recognition and retrieval XV*, volume 6815, page 68150F. International Society for Optics and Photonics, 2008.

[7] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, number CONF, 2011.

[8] Ronan Collobert, Christian Puhrsch, and Gabriel Synnaeve. Wav2letter: an end-to-end convnet-based speech recognition system. *arXiv preprint arXiv:1609.03193*, 2016.

[9] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376. ACM, 2006.

[10] Awni Hannun. Sequence modeling with ctc. *Distill*, 2017. doi: 10.23915/distill.00008. `https://distill.pub/2017/ctc`.

[11] Awni Hannun, Vineel Pratap, Jacob Kahn, and Wei-Ning Hsu. Differentiable weighted finite-state transducers. *arXiv preprint arXiv:2010.01003*, 2020.

[12] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[13] Kevin Knight and Jonathan May. Applications of weighted automata in natural language processing. In *Handbook of Weighted Automata*, pages 571–596. Springer, 2009.

[14] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational linguistics*, 23(2):269–311, 1997.

[15] Mehryar Mohri. Weighted automata algorithms. In *Handbook of weighted automata*, pages 213–254. Springer, 2009.

[16] Mehryar Mohri, Fernando Pereira, and Michael Riley. The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231(1):17–32, 2000.

[17] Mehryar Mohri, Fernando Pereira, and Michael Riley. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88, 2002.

[18] Mehryar Mohri, Fernando Pereira, and Michael Riley. Speech recognition with weighted finite-state transducers. In *Springer Handbook of Speech Processing*, pages 559–584. Springer, 2008.

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[20] FC Pereira and Michael D Riley. Speech recognition by composition of weighted finite automata. *Finite-state language processing*, page 431, 1997.

[21] Fernando Pereira, Michael Riley, and Richard Sproat. Weighted rational transductions and their application to human language processing. In *HUMAN*

LANGUAGE TECHNOLOGY: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994, 1994.

[22] Daniel Povey, Vijayaditya Peddinti, Daniel Galvez, Pegah Ghahremani, Vimal Manohar, Xingyu Na, Yiming Wang, and Sanjeev Khudanpur. Purely sequence-trained neural networks for asr based on lattice-free mmi. 2016.

[23] Richard Sproat, William Gale, Chilin Shih, and Nancy Chang. A stochastic finite-state word-segmentation algorithm for chinese. *Computational Linguistics*, 22(3):377–404, 1996.