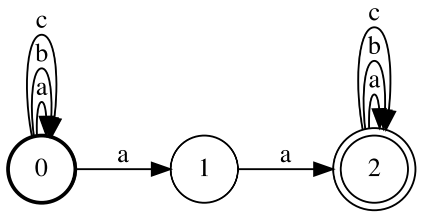

Figure 6.1: A graph which matches the bigram *aa*.

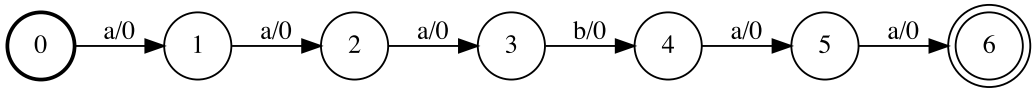

Figure 6.2: A representation of the sequence *aaabaa* as a linear graph for use in computing the frequency of a given n-gram.

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

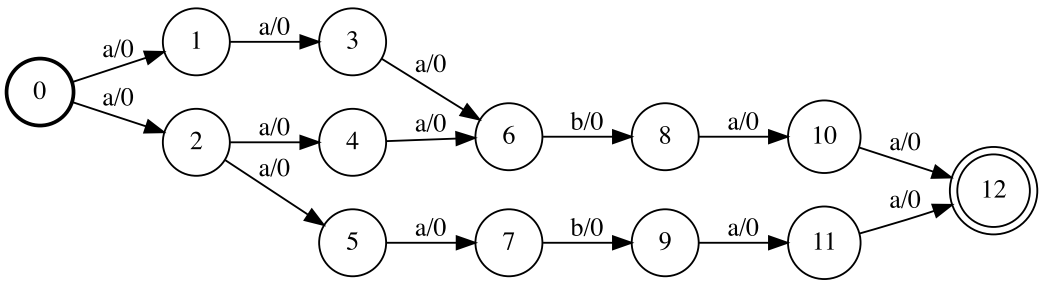

Figure 6.3: The graph represents the intersection of the n-gram graph for *aa* with the graph representing *aaabaa*. The number of unique paths in this graph, in this case 3, is the frequency of the n-gram *ab*.

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.

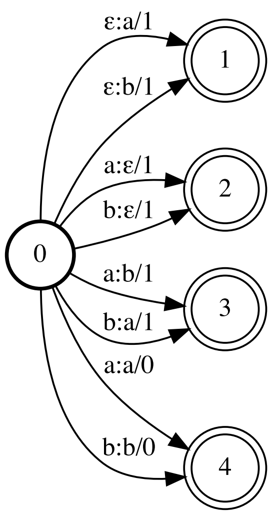

Figure 6.4: An example of an edits graph $\mathcal{E}$ for the token set $\{a, b\}$. The graph encodes the allowed edit distance operations and their associated penalties.

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 $0 \to 1 \to 6 \to 11 \to 16$ 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

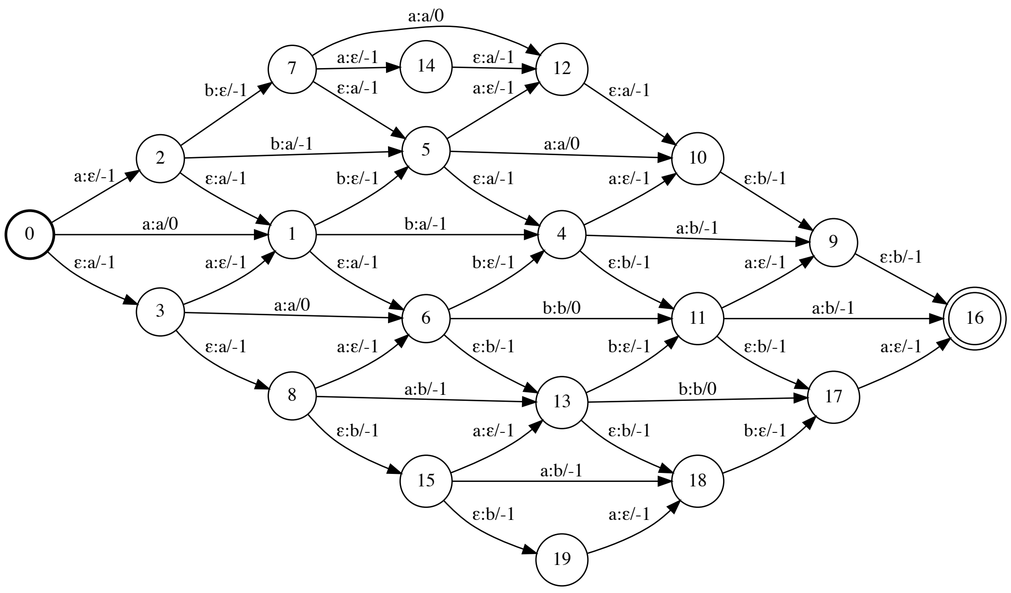

Figure 6.5: The graph $\mathcal{P} = \mathcal{X} \circ \mathcal{E}^* \circ \mathcal{Y}$ represents all possible ways to convert $\mathbf{x} = aba$ to $\mathbf{y} = aabb$ using insertions, deletions, and substitutions. Since the penalties are negative, the score of the highest scoring path is the edit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

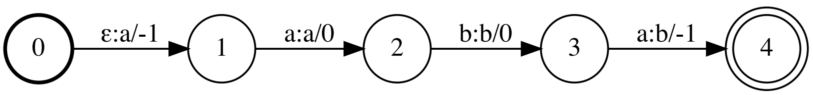

Figure 6.6: The Viterbi path of the graph $\mathcal{P} = \mathcal{X} \circ \mathcal{E}^* \circ \mathcal{Y}$. The edit distance is 2 with one insertions (the arc between states 0 and 1) and one substitution (the arc between states 3 and 4).

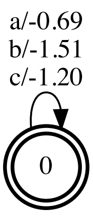

Figure 6.7: A unigram graph $\mathcal{U}$ for $\{a, b, c\}$.

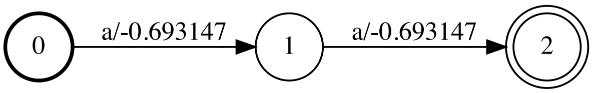

Figure 6.8: The sequence aa scored with the unigram log probabilities from the graph $\mathcal{U}$ in figure 6.7.

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. ■

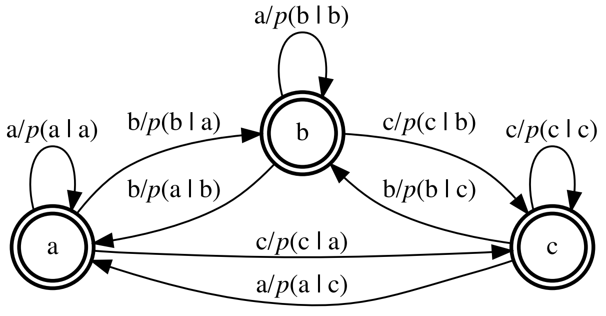

Figure 6.9: A bigram model for the token set $\{a, b, c\}$. Each arc is labeled with the next observed token and the corresponding bigram probability.

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 *abbbba*, *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 $\mathcal{A}_{\mathbf{X},\mathbf{y}}$:

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

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, $\log Z = \mathrm{LSE}(\mathcal{E})$.

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.

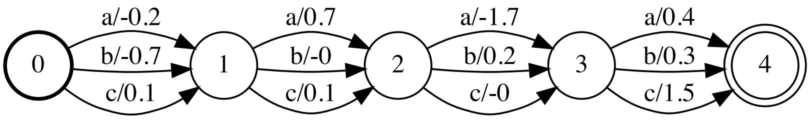

Figure 6.10: An emissions graph $\mathcal{E}$ with $T = 4$ time-steps and a token set of $\{a, b, c\}$.

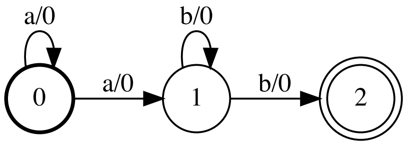

Figure 6.11: The ASG alignment graph $\mathcal{A}_{\mathbf{y}}$ for the sequence ab. The graph encodes the fact that each output token can repeat one or more times in an arbitrary length alignment.

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

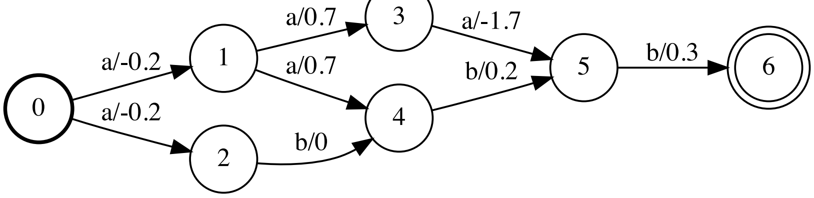

Figure 6.12: The alignment graph $\mathcal{A}_{\mathbf{X},\mathbf{y}}$ for the input $\mathbf{X}$ with $T = 4$ time-steps and an output $\mathbf{y} = ab$.

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 $\mathbf{a}$ 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:

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

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:

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

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:

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

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

$$\underset{x}{\mathrm{LSE}}(x + y) = \mathrm{LSE}(x) + y.$$

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:

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

The first term on the right is:

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

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.

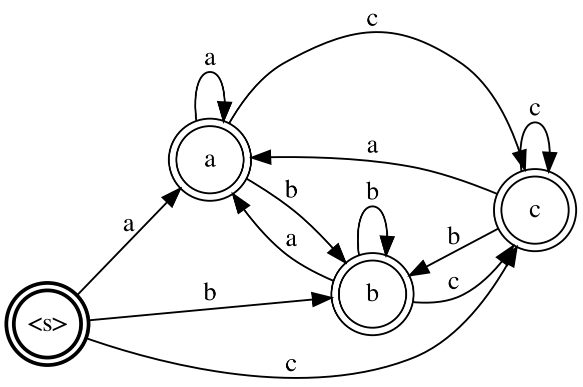

Figure 6.13: The acceptor represents a bigram transition model for the token set $\{a, b, c\}$ (the arc weights are not shown). Each path begins in the start state denoted by the start-of-sequence symbol <s>.

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.

```
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.

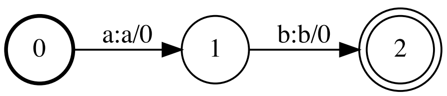

Figure 6.15: The graph $\mathcal{Y}$ corresponding to the target sequence $\mathbf{y} = ab$.

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`

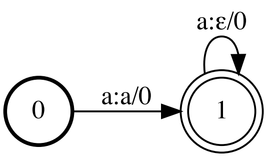

Figure 6.16: An individual token graph $\mathcal{T}_a$ for the token a. The graph encodes the fact that the output a can map to one or more repeats in the alignment.

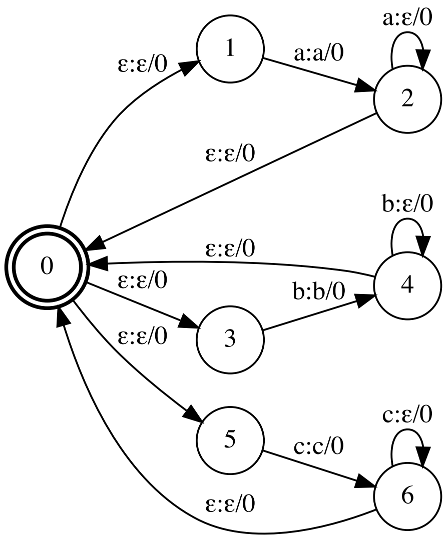

Figure 6.17: The complete token graph $\mathcal{T}$ for the token set $\{a, b, c\}$. The token graph is constructed by taking the closure of the union of the individual token graphs, $\mathcal{T} = (\mathcal{T}_a + \mathcal{T}_b + \mathcal{T}_c)^*$.

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

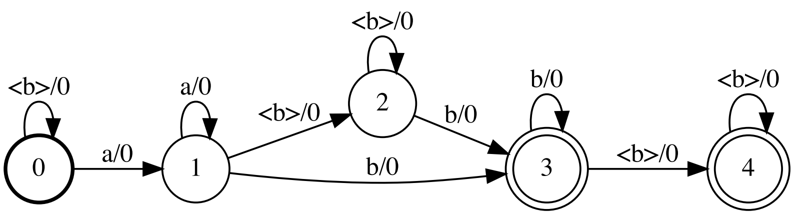

Figure 6.18: The CTC alignment graph $\mathcal{A}_{\mathbf{y}}$ for the sequence *ab*. The graph encodes the fact that each output token can repeat one or more with an optional <b> at the beginning, end, or in between *a* and *b*.

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:

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

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:

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

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

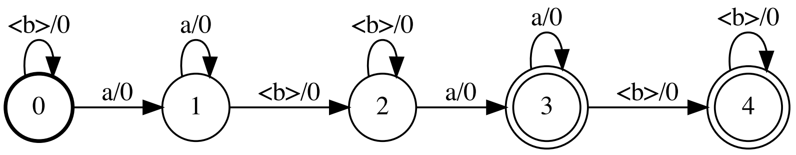

Figure 6.19: The CTC alignment graph $\mathcal{A}_{\mathbf{y}}$ for the sequence aa.

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.