18

Learning Transducers

Pulpo a Feira,
Octopus at a party

Anonymous, From a menu in
O Grove, Galicia

Die Mathematiker sind eine Art
Franzosen: Redet man zu ih-
nen, so übersetzen sie es in ihre
Sprache, und dann ist es alsbald
etwas anderes.

Johann Wolfgang von
Goethe, Maximen und
Reflexionen

18.1 Bilanguages

There are many cases where the function one wants to learn doesn’t just
associate with a given string a label or a probability, but should be able to
return another string, perhaps even written using another alphabet. This is
the case of translation, of course, between two ‘natural’ languages, but also
of situations where the syntax of a text is used to extract some semantics.
And it can be the situation for many other tasks where machine or human
languages intervene.

There are a number of books and articles dealing with machine translation,
but we will only deal here with a very simplified setting consistent with the
types of finite state machines used in the previous chapters; more complex
translation models based on context-free or lexicalised grammars are beyond
the scope of this book.
The goal is therefore to infer special finite transducers, those representing subsequential functions.

### 18.1.1 Rational transducers

Even if in natural language translation tasks the alphabet is often the same for the two languages, this needs not be so. For the sake of generality, we will therefore manipulate two alphabets, typically denoted by Σ for the input alphabet and Γ for the output one.

**Definition 18.1.1 (Transduction)**

A transduction from $\Sigma^*$ to $\Gamma^*$ is a relation $t \subseteq \Sigma^* \times \Gamma^*$.

Even if it is defined as a relation, we choose to give it a direction (from $\Sigma^*$ to $\Gamma^*$) to emphasise the asymmetric nature of the operation. Therefore, transductions are defined by pairs of strings, the first over the input alphabet, and the other over the output alphabet.

A first finite state machine used to recognise transductions is the rational transducer:

**Definition 18.1.2 (Rational transducer)**

A rational transducer is a 5-tuple $T = \langle Q, \Sigma, \Gamma, q_\lambda, E \rangle$:

- $Q$ is a finite set of states,
- $\Sigma, \Gamma$ are the input and output alphabets,
- $q_\lambda \in Q$ is the unique initial state,
- $E \subset (Q \times \Sigma^* \times \Gamma^* \times Q)$ is a finite set of transitions.

Rational transducers can be used like usual finite state machines; they recognise transductions. Given a transducer $T = \langle Q, \Sigma, \Gamma, q_\lambda, E \rangle$, the transduction recognised by $T$, denoted by $t_T$, is the set of all pairs that one can read on a path starting in state $q_\lambda$. If we want to use a different initial state (say $q$), the transduction will be denoted by $t_{T_q}$.

Note that rational transducers have no final states; the parse can halt in any state. Since these are finite state machines, they will be drawn as such. Each transition will be represented by an edge labelled by a pair $x :: y$.

**Definition 18.1.3 (Translation)**

The string $y$ is a translation of the string $x$ if $x = x_1 \cdots x_n$, $y = y_1 \cdots y_n$ ($x_i \in \Sigma^*$, $y_i \in \Gamma^*, \forall i \in [n]$) and there is a sequence of states $q_{i_0}, \ldots, q_{i_n}$ such that $\forall j \in [n], (q_{i_{j-1}}, x_j, y_j, q_{i_j}) \in E$, with $q_{i_0} = q_\lambda$. 

In Figure 18.1 is represented a very simplified rational transducer to translate from French to English. Note that in the above definition the $x_j$ and $y_j$ are substrings, not individual symbols. Notation $x :: y$ gives an idea of determinism that is sometimes desirable and that we will build upon in the next section.

### 18.1.2 Sequential transducers

Following from the earlier definition, the next step consists in reading the input symbols one by one and in a deterministic way:

**Definition 18.1.4 (Sequential transducers)** A **sequential transducer** $T = (Q, \Sigma, \Gamma, q_\lambda, E)$ is a rational transducer such that $E \subset Q \times \Sigma \times \Gamma^* \times Q$ and $\forall (q, a, u, q') \in E \Rightarrow u = v \land q' = q''$.

In the definition above the transition system has become deterministic. This will allow us to do two things:

- associate with $E$ a new function $\tau_E$ such that $\tau_E(q, a) = (v, q')$ for every $(q, a, u, q') \in E$. Let us also associate with $E$ the two projections: $\tau_1 : Q \times \Sigma \to \Gamma^*$ and $\tau_2 : Q \times \Sigma \to Q$, with

  $$(q, a, w, q') \in E \iff \tau_E(q, a) = (w, q'), \tau_1(q, a) = w, \text{ and } \tau_2(q, a) = q'.$$  

  In the same way as with deterministic automata, we can extend $\tau_E$ to a function $Q \times \Sigma \to \Gamma^* \times Q$ and write $\tau_E(q, \lambda) = (\lambda, q)$ and $\tau_E(q, a \cdot u) = (\tau_1(q, a) \cdot \tau_1(\tau_2(q, a), u), \tau_2(\tau_2(q, a), u)).$

- name as usual each state by the shortest prefix over the input alphabet that reaches the state.

**Example 18.1.1** The transducer represented in Figure 18.1 is not sequential because the inputs are strings and not symbols. In this case an alternative sequential transducer can be built, but this is not the general case because of
Learning Transducers

the possible lack of determinism. The finite state machine from Figure 18.2 is a sequential transducer. One has, for instance $\tau_E(q_1, 0101) = (0111, q_\lambda)$.

![Sequential transducer dividing by 3 in base 2.](image)

Fig. 18.2. Sequential transducer dividing by 3 in base 2.

Properties 18.1.1

- The transduction produced by a sequential transducer is a relation $t$ over $\Sigma^* \times \Gamma^*$ that is functional and total, i.e. given any $x \in \Sigma^*$ there is exactly one string $y \in \Gamma^*$ such that $(x, y) \in t$.
  
  With the better adapted functional notation: A transduction $t$ is a total function : $\Sigma^* \rightarrow \Gamma^*$.

- The sequential transductions preserve the prefixes, i.e. $t(\lambda) = \lambda$ and $\forall u, v \in \Sigma^*$, $t(u) \in \text{PREF}(t(uv))$.

We can deduce from the above properties that not all finite transductions can be produced by a sequential transducer. But some computations are possible: For instance in Figure 18.2 we present a transducer that can ‘divide’ by 3 an integer written in base 2. For example, given the input string $u = 100101$ corresponding to 37, the corresponding output is $t(u) = 001100$, which is a binary encoding of 12. The operation is the integer division; one can notice that each state corresponds to the rest of the division by 3 of the input string.

In order to be a bit more general, we introduce subsequential transducers, where a string can be generated at the end of the parse.

18.1.3 Subsequential transducers

We only add to the previous definition a new function $\sigma$, called the state output, which in any state can produce a string when halting in that state.

Definition 18.1.5 (Subsequential transducer) A subsequential transducer is a 6-tuple $(Q, \Sigma, \Gamma, q_\lambda, E, \sigma)$ such that $(Q, \Sigma, \Gamma, q_\lambda, E)$ is a sequential transducer and $\sigma : Q \rightarrow \Gamma^*$ is a total function.
The transduction \( t : \Sigma^* \rightarrow \Gamma^* \) is now defined as \( t(x) = t'(x)\sigma(q) \) where \( t'(x) \) is the transduction produced by the associated sequential transducer and \( q \) is the state reached with the input string \( x \). We again denote by \( t_T(q) \) the transduction realised by the transducer \( T \) using \( q \) as the initial state: \( t_T(q) \subseteq \Sigma^* \times \Gamma^* \).

**Definition 18.1.6 (Subsequential transducer)** A transduction is **subsequential** if \( \text{def} \) there exists a subsequential transducer that recognises it.

Intuitively, a subsequential transduction is one that can be produced from left to right using a bounded amount of memory, in the same way as a regular language is composed of strings recognised by a device reading from left to right and using also bounded memory. But it also corresponds to an optimistic (and thus naive) parse: The associated function has to be total, and thereby, translation on the fly is always possible (since we know it is not going to fail).

**Example 18.1.2** A first example is that of the multiplication by any number in any base. The case where we are in base 2 and we multiply by 3 is represented in Figure 18.3. In this example we have: \( \tau(q_\lambda, 1) = (1, q_1) \), \( \tau_1(q_1, 0) = 1 \), and \( \tau_2(q_1, 1) = q_{11} \).

![Fig. 18.3. Subsequentinal transducer multiplying by 3 in base 2. For input 101, output is 1111.](image)

**Example 18.1.3** Another example is that of the replacement of a pattern by a special string. Here we give the example of pattern \( \text{abaa} \) in Figure 18.4.

![Fig. 18.4. Subsequentinal transducer replacing \( \text{abaa} \).](image)

A third example concerns the translation from numbers written in English to numbers written with the Roman notation. We do not explicit the rather large transducer here, but hope to convince that indeed a bounded memory is sufficient to translate all the numbers less than 5000, like ‘four hundred and seventy three’ into ‘CDLXXIII’.
Fig. 18.4. Transducer replacing in each string the first occurrence of \( \text{abaa} \) by string \( \text{u} \). All other symbols remain identical.

On the other hand, it is easy to show that not all transductions can be recognised by subsequential transducers. A typical example of a transduction that is not subsequential concerns reversing a string. Suppose we would like to consider the transduction containing all pairs of strings \((w, w^R)\) where \(w^R\) is the string \(w\) written in reverse order, i.e. for \(w = \text{abcd}\), \(w^R = \text{dcba}\).
We can use a traditional pumping lemma from formal language theory to prove that this transduction is not subsequential.

18.2 Ostia, a first algorithm that learns transducers

The first algorithm we introduce to learn transducers is called Ostia (Onward Subsequential Transducer Inference Algorithm), a state-merging algorithm based on the same ideas as algorithms Rpni (page 301) or Alergia (page 398). The algorithm builds a special prefix-tree transducer, and from there, through both state merging operations, and advancing the translations as early as possible, a transducer is obtained (in polynomial time), which both is consistent with the learning data and is the correct transducer each time a characteristic sample is contained in the learning sample.

18.2.1 Incomplete transducers

Even if the goal is to learn total functions, we need to be able to denote the fact that in a given state, the information is still unknown. We therefore add to \(\Gamma^*\) a new symbol, \(\bot\), to indicate that the information is still unknown. We denote by \(\hat{\Gamma}^*\) the set \(\Gamma^* \cup \{\bot\}\).

\(\bot\) should be interpreted as the empty set: When searching for a common prefix between \(\bot\) and other strings, \(\bot\) plays no part (it is neutral for the union). On the other hand, if we want to concatenate \(\bot\) with another string,
18.2 Ostia, a first algorithm that learns transducers

⊥ is absorbent. Summarising,

\[ \forall u \in \hat{\Gamma}^* \quad \bot \cdot u = u \cdot \bot = \bot \]

\[ \forall A \in 2^{\hat{\Gamma}^*} \quad \text{lcp}(A \cup \{\bot\}) = \text{lcp}(A). \]

18.2.2 The prefix-tree transducer

Like in the other state merging techniques, the starting point for the algorithms is a tree-like finite state machine, called a prefix-tree transducer (Ptt). There are two steps to build this machine. We use the new symbol, \( \bot \), to indicate that the information is still unknown.

**Algorithm 18.1: Build-Ptt.**

**Data:** a sample \( S \), finite subset of \( \Sigma^* \times \Gamma^* \)

**Result:** a Ptt \( T = \langle Q, \Sigma, \Gamma, q_\lambda, E, \sigma \rangle \)

\( Q \leftarrow \{ q_u : u \in \text{PREF}(\{ x : (x, y) \in S \}) \}; \)

for \( q_u, a \in Q \) do \( \tau(q_u, a) \leftarrow (\lambda, q_u.a) \);

for \( q_u \in Q \) do

\( \text{if } \exists w \in \Gamma^* : (u, w) \in S \text{ then } \sigma(q_u) \leftarrow w \text{ else } \sigma(q_u) \leftarrow \bot \)

end

return \( T \)

**Proposition 18.2.1** Given any finite set of input-output pairs \( S \subset \Sigma^* \times \Gamma^* \), one can build a prefix-tree transducer \( \langle Q, \Sigma, \Gamma, q_\lambda, E, \sigma \rangle \) where:

- \( Q = \{ q_w : (w w', v) \in S \} \),
- \( E = \{ (q_w, a, \lambda, q_w.a) : q_w, q_wa \in Q \} \),
- \( \sigma(q_u) = \{ v \in \hat{\Gamma}^* : (u, v) \in S \} \).

such that the transduction described by \( S \) is generated by it.

The trick is to only translate a string by using the function \( \sigma \). This makes the proof of the above Proposition 18.2.1 straightforward by using Algorithm 18.1.

18.2.3 Advancing: onward transducers

The next important notion is that of advancing: The idea is to privilege translation as soon as possible. This gives us a normal form for transducers.

**Definition 18.2.1 (Onward transducer)** A transducer is onward if\( \text{def} \)

\[ \forall q \in Q, \forall a \in \Sigma, \text{lcp}(\{ u : (q, a, u, q') \in E \} \cup \{ \sigma(q) \}) = \lambda. \]
Remember that \( \text{lcp}(W) \) designs the longest common prefix of set \( W \). We only have to figure out how to deal with value \( \bot \). Since \( \bot \) means that the information is unknown, we always have \( \text{lcp}(\{\bot, u\}) = u \).

This means that the output is assigned to the transitions in such a way as to be produced as soon as we have enough information to do so. For example, the transducer from Figure 18.5(a) is not onward, since some prefixes can be advanced and the transducer of Figure 18.5(b) represents the same transduction, and is this time onward.

**unless** \( \text{lcp}(\{w \in \Gamma^* : \exists x \in \Sigma^* \text{ such that } (x, w) \in t_T \}) = \lambda \), any transducer can be made onward. If this is not the case, this means that all output strings have a common prefix. This prefix can easily be removed before starting and we can therefore suppose that we always are in the situation where \( \text{lcp}(\{w \in \Gamma^* : \exists x \in \Sigma^* \text{ such that } (x, w) \in t_T \}) \neq \lambda \).

### 18.2.4 The onward PTT

Building an onward prefix-tree transducer from a general prefix-tree transducer is easy.

Algorithm **Onward-PTT** takes three arguments: The first is a PTT \( T \), the second is a state \( q \) and the third a string \( f \) such that \( f \) is the longest common prefix of all outputs when starting in state \( q \). When first called in order to make the PTT onward, \( f \) should be of course \( \lambda \) and \( q \) should be set...
18.2 Ostia, a first algorithm that learns transducers

(a) Before advancing. \( \text{lcp}(q_a) = \text{lcp}([\bot, 000, 0]) = 0 \)

(b) After advancing. Now \( \text{lcp}(q_a) = \lambda \).

Fig. 18.6. Making a transducer onward: the case with \( \bot \).

**Algorithm 18.2: Onward-Ptt.**

<table>
<thead>
<tr>
<th>Data: a Ptt : ( T = (Q, \Sigma, \Gamma, q_\lambda, E, \sigma) ), ( q \in Q ), ( u \in \Sigma^* )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result: an equivalent onward Ptt : ( T = (Q, \Sigma, \Gamma, q_\lambda, E, \sigma) ), a string</td>
</tr>
<tr>
<td>( f = \text{lcs}{q_u} )</td>
</tr>
<tr>
<td>for ( a \in \Sigma ) do</td>
</tr>
<tr>
<td>if ( \tau_2(q, a) \in Q ) then ( (T, q, w) \leftarrow \text{ONWARD-Ptt}(T, \tau_2(q, a), a) );</td>
</tr>
<tr>
<td>( \tau_1(q, a) \leftarrow \tau_1(q, a) \cdot w )</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>( f \leftarrow \text{lcp}({\tau_1(q, a)} \cup {\sigma(q)}) );</td>
</tr>
<tr>
<td>if ( f \neq \lambda ) then</td>
</tr>
<tr>
<td>for ( a \in \Sigma ) do ( \tau_1(q, a) \leftarrow f^{-1} \tau_1(q, a) );</td>
</tr>
<tr>
<td>( \sigma(q) \leftarrow f^{-1} \sigma(q) )</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>return ( (T, q, f) )</td>
</tr>
</tbody>
</table>

The value returned is a pair \( (T, f) \). \( T \) is the corresponding onward Ptt rooted in \( u \), \( f \) is the prefix that has been forwarded.
Learning Transducers

18.2.5 A unique onward normal form

Theorem 18.2.2 For any subsequential transduction there exists an onward subsequential transducer with a minimum number of states which is unique up to isomorphisms.

Proof We only sketch the proof, which consists in studying the equivalence relation \(\equiv_T\) over \(\Sigma^*\) based on the transduction \(t_T\), where, if we write \(\text{lcps}(u) = \text{lcp}\{x \in \Gamma^* : \exists y \in \Sigma^* \land (uy, x) \in t_T\}\),

\[u \equiv_T v \iff \forall z \in \Sigma^*, (uz, \text{lcps}(u)u') \in t_T \land (vz, \text{lcps}(v)v') \in t_T \implies u' = v'.\]

In the above \(\text{lcps}(u)\) is a unique string in \(\Gamma^*\) associated with \(u\), corresponding to: Translations of any string starting with \(u\) all start with \(\text{lcps}(u)\). One can then prove that this relation has finite index. Furthermore, a unique (up to isomorphism) subsequential transducer can be built from \(\equiv_T\).

18.3 Ostia

The transducer learning algorithm Ostia (18.7) makes use of Algorithm Ostia-Merge (18.5), which will merge the different states and at the same time, ensure that the result is onward. The merging algorithm is a merge-and-fold variant: It first computes the longest common prefix of every two outputs it is going to have to merge, and then makes the necessary merges.

The first thing we need to be able to check is if two state outputs are identical (or one is \(\perp\)). Formally, we can use Algorithm Ostia-Outputs (18.3) for this.

Algorithm 18.3: Ostia-Outputs.

Input: \(w, w' \in \hat{\Gamma}^*\)

Output:

if \(w = \perp\) then return \(w'\)
else if \(w' = \perp\) then return \(w\)
else if \(w = w'\) then return \(w\)
else return fail

18.3.1 Pushing back

Yet sometimes, we might miss some possible merges because of the onward process. The idea is then that if we can have another symmetrical process
(called *pushing back*) that will somehow differ the outputs, a merge may still be feasible.

We first describe the idea on the simple example represented in Figure 18.7.

(a) Before pushing back. $q_1$ and $q_2$ can’t be merged.

(b) After pushing back on $(q_2, a, 10, q_4)$. String 0 has been pushed back. $q_1$ and $q_2$ can now be merged.

**Fig. 18.7. Pushing back.**

**Algorithm 18.4: Ostia-PushBack.**

**Input:** a transducer $T$, two states $q_1, q_2$, $a \in \Sigma$

**Output:**

$u \leftarrow \text{lcs}\{\tau_1(q_1, a), \tau_1(q_2, a)\}$;

$u_1 \leftarrow u^{-1}\tau_1(q_1, a)$;

$u_2 \leftarrow u^{-1}\tau_1(q_2, a)$;

$\tau_1(q_1, a) \leftarrow u$;

$\tau_1(q_2, a) \leftarrow u$;

for $b \in \Sigma$ do

$\tau_1(\tau_2(q_1, a), b) \leftarrow u_1 \cdot \tau_1(\tau_2(q_1, a), b)$;

$\tau_1(\tau_2(q_2, a), b) \leftarrow u_2 \cdot \tau_1(\tau_2(q_1, a), b)$

end

$\sigma(\tau_2(q_1, a)) \leftarrow u_1 \cdot \sigma(\tau_2(q_1, a))$;

$\sigma(\tau_2(q_2, a)) \leftarrow u_2 \cdot \sigma(\tau_2(q_2, a))$;

return $T$

Typically $\perp$ just absorbs any pushed back suffix. This corresponds to running Algorithm Ostia-PushBack [18.4].

Let us explain a little Algorithm Ostia-PushBack. It takes as input two transitions both labelled by the same symbol $a$ (one starting in state $q_1$ and the other in state $q_2$ and returns a transducer equivalent to the initial one in which the output has been unified. This is done by *pushing back* whatever uncommon suffixes the algorithm finds. There are cases where, due to loops, the result is that we don’t have $\tau_1(q_1, a) = \tau_2(q_2, a)$.
18.3.2 Merging and Folding in Ostia

The idea is to adapt the merge-and-fold technique introduced in Chapter 12 and that we already used in the stochastic setting later (Chapter 16).

We can now write Algorithm Ostia-Merge:

**Algorithm 18.5: Ostia-Merge.**

**Input:** a transducer $T$, two states $q \in \text{Red}$, $q' \in \text{Blue}$

**Output:** $T$ updated

Let $q_f, a$ and $w$ be such that $(q_f, a, w, q') \in E$;

$\tau(q_f, a) \leftarrow (w, q)$;

return Ostia-Fold($T, q, q'$)

This algorithm calls Ostia-Fold:

**Algorithm 18.6: Ostia-Fold.**

**Input:** a transducer $T$, two states $q$ and $q'$

**Output:** $T$ updated, where subtree in $q'$ is folded into $q$

$w \leftarrow \text{Ostia-Outputs}(\sigma(q), \sigma(q'))$;

if $w = \text{fail}$ then

\begin{verbatim}
return fail
\end{verbatim}

else

$\sigma(q) \leftarrow w$;

for $a \in \Sigma$ do

if $\tau(q', a)$ is defined then

if $\tau(q, a)$ is defined then

if $\tau_1(q, a) \neq \tau_1(q', a)$ then /* due to loops */

\begin{verbatim}
return fail
\end{verbatim}

else

$T \leftarrow \text{Ostia-PushBack}(T, q, q', a)$;

$T \leftarrow \text{Ostia-Fold}(T, \tau_2(q, a), \tau_2(q', a))$

end

else

$\tau(q, a) \leftarrow \tau(q', a)$

end

end

return $T$

end
Example 18.3.1 Let us run this merge-and-fold procedure on a simple example. Consider the transducer represented in Figure 18.8. Suppose we want to merge states $q_{aa}$ with $q_{\lambda}$. Notice that $q_{aa}$ is the root of a tree.

We first redirect the edge $(q_a, a, 1, q_{aa})$, which becomes $(q_a, a, 1, q_{\lambda})$ (Figure 18.9).

We now can fold $q_{aa}$ into $q_{\lambda}$. This leads to pushing back the second 1 on the edge $(q_{aa}, a, 1, q_{aaa})$. The resulting situation is represented in Figure 18.10.

Then (Figure 18.11) $q_{aaa}$ is folded into $q_a$, and finally $q_{aab}$ into $q_b$. The result is represented in Figure 18.12.
18.3.3 Properties of the algorithm

Algorithm Ostia works as follows: First a PTT is built. Then it is made onward by running algorithm Onward-PTT. There are two nested loops. The outer loop visits all Blue states, whereas the inner loop visits the Red states to try and find a compatible state. The algorithm halts when there are only Red states left.

At each iteration, a Blue state is chosen and compared with each of the Red states. If no merge is possible, the Blue state is promoted to Red and all its successors become Blue.

Properties 18.3.1

- Algorithm Ostia identifies in the limit any (total) subsequential transduction;
- The complexity is $O(n^2(m + |\Sigma|) + nm|\Sigma|)$ where
  - $n$ is the sum of the input string lengths,
  - $m$ is the length of the longest output string.
Algorithm 18.7: Ostia.

**Data:** a sample $S \in \Sigma^* \times \Gamma^*$

**Result:** $T$

$T \leftarrow \text{Onward-Ptt} \left( \text{Build-Ptt}(S) \right)$;

$\text{Red} \leftarrow \{q_\lambda\}$;

$\text{Blue} \leftarrow \{q_a : (au, v) \in S\}$;

while $\text{Blue} \neq \emptyset$ do

choose $q$ in $\text{Blue}$;

if $\exists p \in \text{Red} : \text{Ostia-Merge}(T, p, q) \neq \text{fail}$ then

$T \leftarrow \text{Ostia-Merge}(T, p, q)$

else

$\text{Red} \leftarrow \text{Red} \cup \{q\}$

end

$\text{Blue} \leftarrow \{p : (q, a, v, p) \in E, q \in \text{Red}\} \setminus \text{Red}$

end

return $T$

We do not detail the proof here, which would follow closely the one for RPN1. We only sketch the main elements:

- Identification in the limit is ensured as soon as a characteristic sample is given, which avoids all the inconvenient merges.
- The complexity bound is estimated very broadly. In practice it is much less.

### 18.3.4 A run of algorithm Ostia

Suppose we want to identify the transducer represented in Figure 18.13 using Ostia. The target transducer takes as input any sequence of symbols from $\Sigma = \{a, b\}$, and replaces each $a$ which is not followed by another $a$, by a 0. If not $a$ becomes 0 and $b$ becomes 1.

```
\begin{center}
\begin{tikzpicture}
  \node [state] (q0) at (0,0) {$q_a : \lambda$};
  \node [state] (q1) at (1,0) {$q_a : 1$};
  \node [state] (q2) at (0,-1) {$qa : \lambda$};
  \node [state] (q3) at (1,-1) {b :: 01};
  \node [state] (q4) at (0,-2) {a :: 0};
  \node [state] (q5) at (1,-2) {b :: 1};
  \path
    (q0) edge [loop above] node {a :: 0} (q0)
    (q0) edge [loop above] node {b :: 1} (q1)
    (q1) edge [bend left] node {a :: 0} (q0)
    (q2) edge [loop below] node {b :: 0} (q2)
    (q2) edge [bend left] node {a :: \lambda} (q1)
    (q3) edge [bend left] node {b :: 01} (q2)
    (q4) edge [bend left] node {a :: \lambda} (q1)
    (q4) edge [bend left] node {b :: 1} (q5)
    (q5) edge [bend left] node {a :: 0} (q4)
    (q5) edge [bend left] node {b :: 1} (q3)
\end{tikzpicture}
\end{center}
```

Fig. 18.13. The target.

Typical examples (which make the learning sample), are $(a, 1)$, $(b, 1)$, $(aa, 01)$, $(aaa, 001)$, $(abab, 0101)$. 
We first use Algorithm Build-Ptt (18.1) and build the corresponding Ptt represented in Figure 18.14. Algorithm Onward-Ptt (18.2) is then called and we obtain the onward Ptt (Figure 18.15).

The first merge that we test is between states \( q_a \) (which is Blue) and the unique Red state \( q_\lambda \); the merge is rejected because the finals \( \lambda \) and 1 are different, so the states cannot be merged together. So \( q_a \) is promoted.

Ostia then tries to merge \( q_b \) with \( q_\lambda \); this merge is accepted, and the new transducer is depicted in Figure 18.16.

The next attempt is to merge \( q_{aa} \) with \( q_\lambda \); this merge is rejected again.
18.4 Identifying partial functions

It is easy to see that the Ostia algorithm cannot identify partial functions, i.e. those that would not have translations for each input string: For that just take a function that for a given regular language $L$ translates $w$ into $1$ if $w \in L$, and has no translation for strings outside $L$. If we could identify these functions the algorithm would also be capable of identifying regular languages from text, which is impossible (see Theorem 7.2.3, page 173).

Yet if one wants to use a transducer for a problem of morphology or of automatic translation, it is clear that total functions make little sense: Not
Learning Transducers

every random sequence of symbols should be translated. Furthermore, the absence of some input strings should be used to the advantage of the learning algorithm. As it is, this is not the case: The absence of an input string will mean the absence of an element forbidding a merge, in which case the (bad) merge will be made.

In order to hope to learn such partial functions we therefore need some additional information. This can be of different sorts:

- using negative samples,
- using knowledge about the domain, or about the range of the function.

### 18.4.1 Using negative samples

We just explain the idea. The algorithm is given two samples: One contains transductions, \( \text{i.e.} \) pairs of strings in \( \Sigma^* \times \Gamma^* \), and the second one contains strings from \( \Sigma^* \) that do not admit translation. The algorithm is adapted so that, when checking if a merge is possible, we should check if one of the prohibited strings has not obtained a translation. If so, the merge is rejected. Hence, more possibilities of rejecting strings exist.

The definition of characteristic sample can clearly be adapted to this setting.

### 18.4.2 Using domain knowledge

In Algorithm Ostia (18.7) the function must be total. For straightforward reasons (see Exercise 18.2) it is impossible to identify in the limit partial functions. But if we are given the domain of the partial function then this can be used as background or expert knowledge in the learning function. Moreover, if the domain is a regular language, then the transducer has to respect in some way the structure of the language. This can be used during the learning phase: Indeed, when seeking to merge two states, not only should the transductions correspond (or not be inconsistent), but also should the types of the two states coincide.

**Example 18.4.1** Let us consider the subsequential transducer represented in Figure 18.18(a). Let us suppose that the learning sample contains strings \((a, 1), (b, 1), (abab, 0101)\) and the information that the domain of the function is \((ab + b)^* (a + \lambda)^*\).

Then, the DFA recognising the domain of the language can be represented as in Figure 18.18(b). The initial point of the adapted algorithm consists in labelling the PTT with the indexes of the DFA corresponding to the domain.
18.5 Exercises

18.1 Build the PTT for \{ (abba, abba), (abaaa, aa), (bbaba, bba), (aa, aa) \}.

18.2 Prove that learning partial subsequential transducers is hard: They cannot be identified in the limit. Hint: consider transducers that translate every string into \( \lambda \).

18.3 Run Ostia on the following sample \{ (aa, a), (aaa, b), (baa, a) \}.

18.4 Suppose we know that the domain of the function contains only those strings of length at most 4. What is learnt?

18.5 Run Ostia on the PTT from Figure 18.19, then run the adapted version using the domain information.
18.6 Conclusions of the chapter and further reading

18.6.1 Bibliographical background

Transducers were introduced by Marcel-Paul Schützenberger and studied by a number of authors since then [RS91, RS95], with Jean Berstel’s book being the reference [Ber79]. Mehryar Mohri has been advocating similarities between the different types of finite state machines, and has been defending the point of view that transducers represent the initial object, in the sense that a DFA (or NFA) can be seen as a machine for a transduction over \( \Sigma^* \times \{0, 1\} \), multiplicity automata as machines for transductions over \( \Sigma^* \times \mathbb{R} \) and PFA do the same over \( \Sigma^* \times (\mathbb{R} \cap [0; 1]) \) [Moh97]. Applications of transducers to natural language processing are still unclear, some specialists believing the mechanisms of a transducer to be too poor to express the subtleties of language. Conversely, Brian Roark and Richard Sproat [RS07] argue that nearly all morphological rules can be described by finite state transducers. Applications to machine translation were done by Enrique Vidal, Francisco Casacuberta and their team [ABC+01, CV04].

The algorithm Ostia that we describe in this chapter was designed by Jose Oncina, Pedro García and Enrique Vidal [OGV93]. The different extensions described in Section 18.4 are called Ostia-N for the one that uses negative examples and Ostia-D for the algorithm that makes use of domain knowledge. The domain version (Algorithm 18.4.2, page 454) was introduced by Jose Oncina and Miguel Angel Varó [OV96]. Similar ideas were explored later by Christopher Kermorvant et al. [KdlH02, CFKdlH04, KdlHD04] in more general grammatical inference settings under the name of “learning with help”.

A version with queries of Ostia was written by Juan Miguel Vilar [Vil96]. Making Ostia practical has been an important issue. Using dictionaries and word alignments has been tested [Vil00]. This also allowed to add probabilities to the transducers.

Theoretical results concerning stochastic transducers can be found in [CdhH00]: Some decoding problems are proved to be \( \mathcal{NP} \)-hard, but these hold specifically in the non-deterministic setting. Results follow typically from results concerning probabilistic automata.

Between the hardest attempts to learn difficult transducers, Alex Clark won the Tenjinno competition in 2006 by using Ostia and several other ideas [Cla06].
18.6 Conclusions of the chapter and further reading

18.6.2 Some alternative lines of research

There are of course many alternative approaches than building finite state machines for translation tasks.

Learning probabilistic transducers is an important topic: These can be used to define stochastic edit distances [BJS06]. Tree transducers are also important as they can be used with XML. Another crucial issue is that of smoothing for which little is known. Researchers attempting to learn wrappers need to learn a function that transforms a tree into another, where the important information is made clear [CGLN05].

18.6.3 Open problems and possible new lines of research

Machine translation tools are going to be increasingly important over the next years. But as shown during the Tenjinno competition, effort has to be made in many directions if transducer learning can be successfully used in applications.

An important question that deserves more attention corresponds to extending the definitions and algorithms to the probabilistic case.