Neural Finite State Transducers: Beyond Rational Relations

Chu-Cheng Lin
Computer Science Department
Johns Hopkins University
Baltimore, MD 21218, USA
kitsing@cs.jhu.edu

Hao Zhu
Dept. of Computer Science and Technology
Tsinghua University
Beijing, China
zhuhao15@mails.tsinghua.edu.cn

Matthew R. Gormley
Machine Learning Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA
mgormley@cs.cmu.edu

Jason Eisner
Computer Science Department
Johns Hopkins University
Baltimore, MD 21218, USA
jason@cs.jhu.edu

Abstract

We introduce neural finite state transducers (NFSTs), a family of string transduction models defining joint and conditional probability distributions over pairs of strings. The probability of a string pair is obtained by marginalizing over all its accepting paths in a finite state transducer. In contrast to ordinary weighted FSTs, however, each path is scored using an arbitrary function such as a recurrent neural network, which breaks the usual conditional independence assumption (Markov property). NFSTs are more powerful than previous finite-state models with neural features (Rastogi et al., 2016). We present training and inference algorithms for locally and globally normalized variants of NFSTs. In experiments on different transduction tasks, they compete favorably against seq2seq models while offering interpretable paths that correspond to hard monotonic alignments.

1 Introduction

Weighted finite state transducers (WFSTs) have been used for decades to analyze, align, and transduce strings in language and speech processing (Roche and Schabes, 1997; Mohri et al., 2008). They form a family of efficient, interpretable models with well-studied theory. A WFST describes a function that maps each string pair $(\mathbf{x}, \mathbf{y})$ to a weight—often a real number representing $p(\mathbf{x}, \mathbf{y})$ or $p(\mathbf{y} \mid \mathbf{x})$. The WFST is a labeled graph, in which each path $\mathbf{a}$ represents a sequence of operations that describes how some $\mathbf{x}$ and some $\mathbf{y}$ could be jointly generated, or how $\mathbf{x}$ could be edited into $\mathbf{y}$. Multiple paths for the same $(\mathbf{x}, \mathbf{y})$ pair correspond to different analyses (labeled alignments) of that pair.

However, WFSTs can only model certain functions, known as the rational relations (Berstel and Reutenauer, 1988).The weight of a path is simply the product of the weights on its arcs. This means that in a random path of the form $a \rightsquigarrow b \rightsquigarrow c$, the two subpaths $\rightsquigarrow$ are conditionally independent given their common state b: a Markov property.

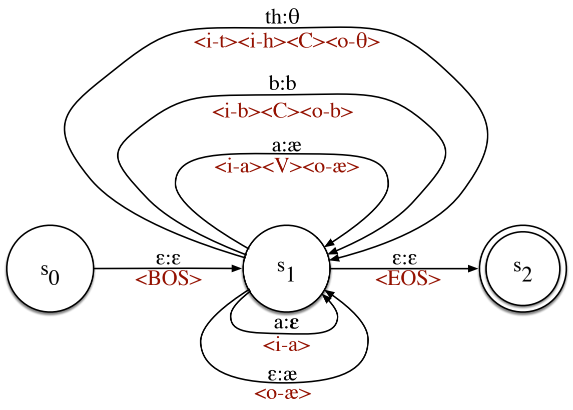

Figure 1: A marked finite-state transducer $\mathcal{T}$. Each arc in $\mathcal{T}$ is associated with input and output substrings, listed above the arcs in the figure. The arcs are not labeled with weights as in WFSTs. Rather, each arc is labeled with a sequence of *marks* (shown in brown) that featurize its qualities. The neural scoring model scores a path by scoring each mark in the context of all marks on the entire path. The example shown here is from the G2P application of §4.1; for space, only a few arcs are shown. ε represents the empty string.

In this paper, we propose neural finite state transducers (NFSTs), in which the weight of each path is instead given by some sort of neural network, such as an RNN. Thus, the weight of an arc can depend on the context in which the arc is used. By abandoning the Markov property, we lose exact dynamic programming algorithms, but we gain expressivity: the neural network can capture dependencies among the operations along a path. For example, the RNN might give higher weight to a path if it is "internally consistent": it might thus prefer to transcribe a speaker's utterance with a path that maps similar sounds in similar contexts to similar phonemes, thereby adapting to the speaker's accent.

Consider a finite-state transducer $\mathcal{T}$ as in Figure 1 (see Appendix A for background). Using the composition operator $\circ$, we can obtain a new FST, $\mathbf{x} \circ \mathcal{T}$, whose accepting paths correspond to the accepting paths of $\mathcal{T}$ that have input string $\mathbf{x}$. Similarly, the accepting paths of $\mathcal{T} \circ \mathbf{y}$ correspond to the accepting paths of $\mathcal{T}$ that have output string $\mathbf{y}$. Finally, $\mathbf{x} \circ T \circ \mathbf{y}$ extracts the paths that have both properties. We define a joint probability distribution over $(\mathbf{x}, \mathbf{y})$ pairs by marginalizing over those paths:

$$p(\mathbf{x}, \mathbf{y}) = \sum_{\mathbf{a} \in \mathbf{x} \circ \mathcal{T} \circ \mathbf{y}} p(\mathbf{a}) = \frac{1}{Z(\mathcal{T})} \sum_{\mathbf{a} \in \mathbf{x} \circ \mathcal{T} \circ \mathbf{y}} \tilde{p}(\mathbf{a}) \quad (1)$$

where $\tilde{p}(\mathbf{a})$ is the weight of path $\mathbf{a}$ and $Z(\mathcal{T}) = \sum_{\mathbf{a} \in \mathcal{T}} \tilde{p}(\mathbf{a})$ is a normalization constant.

We define $\tilde{p}(\mathbf{a}) \triangleq \exp G_{\boldsymbol{\theta}}(\mathbf{a})$ with $G_{\boldsymbol{\theta}}(\mathbf{a})$ being some parametric scoring function. In our experiments, we will adopt a fairly simple left-to-right RNN architecture (§2.2), but one could easily substitute fancier architectures. We will also consider defining $G_{\boldsymbol{\theta}}$ by a *locally normalized* RNN that ensures $Z(\mathcal{T}) = 1$.

In short, we use the finite-state transducer $\mathcal{T}$ to compactly define a set of *possible* paths $\mathbf{a}$. The number of paths may be exponential in the size of $\mathcal{T}$, or infinite if $\mathcal{T}$ is cyclic. However, in contrast to WFSTs, we abandon this combinatorial structure in favor of neural nets when defining the probability distribution over $\mathbf{a}$. In the resulting marginal distribution $p(\mathbf{x}, \mathbf{y})$ given in equation (1), the path $\mathbf{a}$ that *aligns* $\mathbf{x}$ and $\mathbf{y}$ is a latent variable. This is also true of the resulting conditional distribution $p(\mathbf{y} \mid \mathbf{x})$.

We explore training and inference algorithms for various classes of NFST models (§3). Classical WFSTs (Mohri et al., 2008) and BiRNN-WFSTs (Rastogi et al., 2016) use restricted scoring functions and so admit exact dynamic programming algorithms. For general NFSTs, however, we must resort to approximate computation of the model's training gradient, marginal probabilities, and predictions. In this paper, we will use sequential importance sampling methods (Lin and Eisner, 2018), leaving variational approximation methods to future work.

Defining models using FSTs has several benefits:

Output-sensitive encoding Currently popular models of $p(\mathbf{y} \mid \mathbf{x})$ used in machine translation and morphology include seq2seq (Sutskever et al., 2014), seq2seq with attention (Bahdanau et al., 2015; Luong et al., 2015), the Transformer (Vaswani et al., 2017). These models first encode $\mathbf{x}$ as a vector or sequence of vectors, and then condition the generation of $\mathbf{y}$ on this encoding. The vector is determined from $\mathbf{x}$ only. This is also the case in the BiRNN-WFST (Rastogi et al., 2016), a previous finite-state model to which we compare. By contrast, in our NFST, the state of the RNN as it reads and transduces the second half of $\mathbf{x}$ is influenced by the first halves of both $\mathbf{x}$ *and* $\mathbf{y}$ and their alignment.

Inductive bias Typically, a FST is constructed with domain knowledge (possibly by compiling a regular expression), so that its states reflect interpretable properties such as syllable boundaries or linguistic features. Indeed, we will show below how to make these properties explicit by "marking" the FST arcs. The NFST's path scoring function then sees these marks and can learn to take them into account. The NFST also inherits any hard constraints from the FST: if the FST omits all $(\mathbf{x}, \mathbf{y})$ paths for some "illegal" $\mathbf{x}, \mathbf{y}$, then $p(\mathbf{x}, \mathbf{y}) = 0$ for any parameter vector $\boldsymbol{\theta}$ (a "structural zero").

Interpretability Like a WFST, an NFST can "explain" why it mapped $\mathbf{x}$ to $\mathbf{y}$ in terms of a latent path $\mathbf{a}$, which specifies a hard monotonic labeled alignment. The posterior distribution $p(\mathbf{a} \mid \mathbf{x}, \mathbf{y})$ specifies which paths $\mathbf{a}$ are the best explanations (e.g., Table 5).

We conduct experiments on three tasks: grapheme-to-phoneme, phoneme-to-grapheme, and action-to-command (Bastings et al., 2018). Our results on these datasets show that our best models can improve over neural seq2seq and previously proposed hard alignment models.

2 Neuralizing Finite-State Transducers

2.1 Neuralized FSTs

An NFST is a pair $(\mathcal{T}, G_{\boldsymbol{\theta}})$, where $\mathcal{T}$ is an unweighted FST with accepting paths $\mathcal{A}$ and $G_{\boldsymbol{\theta}} : \mathcal{A} \to \mathbb{R}$ is a function that **scores** these paths. As explained earlier, we then refer to $\tilde{p}(\mathbf{a}) = \exp G_{\boldsymbol{\theta}}(\mathbf{a})$ as the **weight** of path $\mathbf{a} \in \mathcal{A}$. A **weighted relation** between input and output strings is given by $\tilde{p}(\mathbf{x}, \mathbf{y})$, which is defined to be the total weight of all paths with input string $\mathbf{x} \in \Sigma^*$ and output string $\mathbf{y} \in \Delta^*$, where where Σ and Δ are the input and output alphabets of $\mathcal{T}$. The real parameter vector $\boldsymbol{\theta}$ can be adjusted to obtain different weighted relations. We

Model	Training Algorithms	Long-Term Output-Output Dependency	Left-to-Right Factorization
WFSTs	Dynamic Programming	✗	✓
BiRNN-WFSTs	Dynamic Programming	✗	✓
Local NFSTs	Importance Sampling	✓	✓
Global NFSTs	Importance Sampling	✓	✗

Table 1: Comparison between WFSTs, BiRNN-WFSTs (Rastogi et al., 2016), and NFSTs.

can normalize $\tilde{p}$ to get a probability distribution as shown in equation (1).

2.2 A basic scoring architecture

Weighted FST. A WFST over the $(+, \times)$ semiring can be regarded as the special case in which $G_{\boldsymbol{\theta}}(\mathbf{a}) \triangleq \sum_{t=1}^{|\mathbf{a}|} g_{\boldsymbol{\theta}}(a_t)$. This is a sum of scores assigned to the arcs in $\mathbf{a} = a_1 a_2 \cdots$.

Marked FST. Our innovation is to allow the arcs' scores to depend on their context in the path. Now $\boldsymbol{\theta}$ no longer associates a fixed score with each arc. Rather, we assume that each arc a in the FST comes labeled with a sequence of **marks** from a **mark alphabet** Ω, as illustrated in Figure 1. The marks reflect the FST constructor's domain knowledge about what arc a does (see §4.2 below). We now define $G_{\boldsymbol{\theta}}(\mathbf{a}) = G_{\boldsymbol{\theta}}(\boldsymbol{\omega}(\mathbf{a}))$, where $\boldsymbol{\omega}(\mathbf{a}) = \boldsymbol{\omega}(a_1)\boldsymbol{\omega}(a_2) \cdots \in \Omega^*$ is the concatenated sequence of marks from the arcs along path $\mathbf{a}$.

It is sometimes helpful to divide marks into different classes. An arc can be regarded as a possible "edit" that aligns an input substring with an output substring in the context of transitioning from one FST state to another. The arc's **input marks** describe its input substring, its **output marks** describe its output substring, and the remaining marks may describe other properties of the arc's aligned input-output pair or the states that it connects.

Recall that an FST encodes domain knowledge. Its paths represent alignments between input and output strings, where each alignment specifies a segmentation of $\mathbf{x}$ and $\mathbf{y}$ into substrings labeled with FST states. Decorating the arcs with marks furnishes the path scoring model with domain-specific information about the alignments.

RNN scoring. If $\boldsymbol{\theta}$ merely associated a fixed score with each mark, then the marked FST would be no more powerful than the WFST. To obtain contextual mark scores as desired, one simple architecture is a recurrent neural network:

$$G_{\boldsymbol{\theta}}(\boldsymbol{\omega}) \triangleq \sum_{t=1}^{|\boldsymbol{\omega}|} g_{\boldsymbol{\theta}}(\mathbf{s}_{t-1}, \omega_t) \tag{2}$$

$$\mathbf{s}_t = f_{\boldsymbol{\theta}}(\mathbf{s}_{t-1}, \omega_t), \text{ with } \mathbf{s}_0 = \mathbf{0} \tag{3}$$

where $\mathbf{s}_{t-1} \in \mathbb{R}^d$ is the hidden state vector of the network after reading $\omega_1 \cdots \omega_{t-1}$. The $g_{\boldsymbol{\theta}}$ function defines the score of reading ω_t in this left context, and $f_{\boldsymbol{\theta}}$ defines how doing so updates the state.

In our experiments, we chose $f_{\boldsymbol{\theta}}$ to be the GRU state update function (Cho et al., 2014). We defined $g_{\boldsymbol{\theta}}(\mathbf{s}, \omega_t) \triangleq (\mathbf{Ws} + \mathbf{b})^\top \text{emb}(\omega_t)$. The parameter vector $\boldsymbol{\theta}$ specifies the GRU parameters, $\mathbf{W}$, $\mathbf{b}$, and the mark embeddings $\text{emb}(\omega)$.

One could easily substitute much fancier architectures, such as a stacked BiLSTM with attention (Tilk and Alumäe, 2016), or a Transformer (Vaswani et al., 2017).

2.3 Partitioned hidden vectors

In hopes of improving the inductive bias of the learner, we partitioned the hidden state vector into three sub-vectors: $\mathbf{s}_t = [\mathbf{s}_t^{\mathrm{a}}; \mathbf{s}_t^{\mathrm{x}}; \mathbf{s}_t^{\mathrm{y}}]$. The mark scoring function $f_{\boldsymbol{\theta}}(\mathbf{s}_{t-1}, \omega_t)$ was as before, but we restricted the form of $g_{\boldsymbol{\theta}}$, the state update function. $\mathbf{s}_t^{\mathrm{a}}$ encodes *all* past marks and depends on the *full* hidden state so far: $\mathbf{s}_t^{\mathrm{a}} = g_{\boldsymbol{\theta}}^{\mathrm{a}}(\mathbf{s}_{t-1}, \omega_t)$. However, we make $\mathbf{s}_t^{\mathrm{x}}$ encode only the sequence of past *input* marks, ignoring all others. Thus, $\mathbf{s}_t^{\mathrm{x}} = g_{\boldsymbol{\theta}}^{\mathrm{x}}(\mathbf{s}_{t-1}^{\mathrm{x}}, \omega_t)$ if ω_t is an input mark, and $\mathbf{s}_t^{\mathrm{x}} = \mathbf{s}_{t-1}^{\mathrm{x}}$ otherwise. Symmetrically, $\mathbf{s}_t^{\mathrm{y}}$ encodes only the sequence of past *output* marks. This architecture is somewhat like Dyer et al. (2016), which also uses different sub-vectors to keep track of different aspects of the history.

2.4 Local normalization

A difficulty with the general model form in equation (1) is that the normalizing constant $Z(\mathcal{T}) = \sum_{\mathbf{a} \in \mathcal{T}} \tilde{p}(\mathbf{a})$ must sum over a large set of paths—in fact, an infinite set if $\mathcal{T}$ is cyclic. This sum may diverge for some values of the parameter vector $\boldsymbol{\theta}$, which complicates training of the model (Dreyer, 2011). Even if the sum is known to converge, it is in general intractable to compute it exactly. Thus, estimating the gradient of $Z(\mathcal{T})$ during training involves approximate sampling from the typically high-entropy distribution $p(\mathbf{a})$. The resulting estimates are error-prone because the sample size tends to be too small and the approximate sampler is biased.

A standard solution in the WFST setting (e.g. Cotterell et al., 2014) is to use a locally normalized model, in which $Z(\mathcal{T})$ is guaranteed to be 1.[1] The big summation over all paths $\mathbf{a}$ is replaced by small summations—which can be computed explicitly—over just the outgoing edges from a given state.

Formally, we define the unnormalized score of arc a_i in the context of path $\mathbf{a}$ in the obvious way, by summing over the contextual scores of its marks:

$$\tilde{g}_{\boldsymbol{\theta}}(a_i) \triangleq \sum_{t=j+1}^{k} g_{\boldsymbol{\theta}}(\mathbf{s}_{t-1}, \omega_t) \tag{4}$$

where $j = |\boldsymbol{\omega}(a_1)\cdots\boldsymbol{\omega}(a_{i-1})|$ and $k = |\boldsymbol{\omega}(a_1)\cdots\boldsymbol{\omega}(a_i)|$. Its normalized score is then

$$g_{\boldsymbol{\theta},\mathcal{T}}(a_i) \triangleq \log\left(\exp \tilde{g}_{\boldsymbol{\theta}}(a_i) / \sum_{a'} \exp \tilde{g}_{\boldsymbol{\theta}}(a')\right)$$

where a' ranges over all arcs in $\mathcal{T}$ (including a_i itself) that emerge from the same state as a_i does. We can now score the paths in $\mathcal{T}$ using

$$G_{\boldsymbol{\theta},\mathcal{T}}(\mathbf{a}) = \sum_{i=1}^{|\mathbf{a}|} g_{\boldsymbol{\theta},\mathcal{T}}(a_i) \tag{5}$$

This gives rise to a proper probability distribution $p(\mathbf{a}) \triangleq \tilde{p}(\mathbf{a}) = \exp G_{\boldsymbol{\theta},\mathcal{T}}(\mathbf{a})$ over the paths of $\mathcal{T}$. No global normalization constant is necessary. However, note that the scoring function now requires $\mathcal{T}$ as an extra subscript, because it is necessary when scoring $\mathbf{a}$ to identify the competitors in $\mathcal{T}$ of each arc a_i. Thus, when $p(\mathbf{x}, \mathbf{y})$ is found as usual by summing up the probabilities of all paths in $\mathbf{x} \circ \mathcal{T} \circ \mathbf{y}$, each path is still scored using its arcs' competitors from $\mathcal{T}$. This means that each state in $\mathbf{x} \circ T \circ \mathbf{y}$ must record the state in $\mathcal{T}$ from which it was derived.

3 Sampling, Training, and Decoding

3.1 Sampling from conditioned distributions with amortized inference

Many algorithms for working with probability distributions—including our training and decoding algorithms below—rely on conditional sampling. In general, we would like to sample a path of $\mathcal{T}$ given the knowledge that its input and output strings fall into sets X and Y respectively.[2] If X and Y are regular languages, this is equivalent to defining $\mathcal{T}' = X \circ \mathcal{T} \circ Y$ and sampling from

$$p(\mathbf{a} \mid \mathcal{T}') \triangleq \frac{\tilde{p}(\mathbf{a})}{\sum_{\mathbf{a}' \in \mathcal{T}'} \tilde{p}(\mathbf{a}')}, \tag{6}$$

Due to the nonlinearity of $G_{\boldsymbol{\theta}}$, the denominator of equation (6) is generally intractable. If $\mathcal{T}'$ is cyclic, it cannot even be computed by brute-force enumeration. Thus, we fall back on normalized importance sampling, directly adopting the ideas of Lin and Eisner (2018) in our more general FST setting. We employ a proposal distribution q:

$$\begin{aligned} p(\mathbf{a} \mid \mathcal{T}') &= \mathbb{E}_{\mathbf{a} \sim q}\left[\frac{p(\mathbf{a} \mid \mathcal{T}')}{q(\mathbf{a})}\right], \quad (7)\\ &\approx \sum_{m=1}^{M} \frac{\tilde{p}(\mathbf{a}^{(m)})}{q(\mathbf{a}^{(m)}) \cdot \hat{Z}} \cdot \mathbb{I}(\mathbf{a} = \mathbf{a}^{(m)}) \\ &= \hat{p}(\mathbf{a} \mid \mathcal{T}'), \end{aligned}$$

where $\hat{Z} = \sum_{m'=1}^{M} \frac{\tilde{p}(\mathbf{a}^{(m')})}{q(\mathbf{a}^{(m')})}$, and q is a locally normalized distribution over paths $\mathbf{a} \in \mathcal{T}'$. In this paper we further parametrize q as

$$q_{\boldsymbol{\phi}}(\mathbf{a}; \mathcal{T}') = \prod_{t=1}^{T} q_t(a_t \mid \mathbf{a}_{1\ldots t-1}; \boldsymbol{\phi}, \mathcal{T}'), \tag{8}$$

$$q_t(a \mid \mathbf{a}_{:t-1}; \boldsymbol{\phi}, \mathcal{T}') \propto \exp(g(\mathbf{s}_{t-1}, a_t; \boldsymbol{\theta}, \mathcal{T}) + C_{\boldsymbol{\phi}}),$$

where $C_{\boldsymbol{\phi}} \triangleq C(\mathbf{s}'_t, X, Y, \boldsymbol{\phi}) \in \mathbb{R}$, $\mathbf{s}'_t \triangleq f(\mathbf{s}_{t-1}, \boldsymbol{\omega}(a))$ is a *compatibility function* that is typically modeled using a neural network. In this paper, one the following three cases are encountered:

- $X = \mathbf{x}$, is a string, and $Y = \Delta^*$: in this case $\mathcal{T}' = \mathbf{x} \circ \mathcal{T}$. We let $C_{\boldsymbol{\phi}} = C^{\mathrm{x}}(\mathbf{s}'_t, \mathrm{RNN}^{\mathrm{x}}(\mathbf{x}, i, \boldsymbol{\phi}); \boldsymbol{\phi})$, where i is the length of the input prefix in $\mathbf{a}_{1\ldots t}.a$, $\mathrm{RNN}^{\mathrm{x}}(\mathbf{x}, i, \boldsymbol{\phi})$ is the hidden state of the i-th position after reading $\mathbf{x}$ (not $\mathbf{a}$ nor $\boldsymbol{\omega}$) backwards, and $C^{\mathrm{x}}(\cdot,\cdot)$ is a feed-forward network that takes the concatenated vector of all arguments, and outputs a real scalar. We describe the parametrization of C^{x} in Appendix C.1.

[1] Provided that every state in $\mathcal{T}$ is co-accessible, i.e., has a path to a final state.

[2] When X or Y is larger than a single string, it is commonly all of Σ^* or Δ^* respectively, in which case conditioning on it gives no information.

- $X = \Sigma^*$, and $Y = \mathbf{y}$ is a string: in this case $\mathcal{T}' = \mathcal{T} \circ \mathbf{y}$. We let $C_{\boldsymbol{\phi}} = C^{\mathrm{y}}(\mathbf{s}'_t, \mathrm{RNN}^{\mathrm{y}}(\mathbf{y}, j, \boldsymbol{\phi}); \boldsymbol{\phi})$, where j is the length of the output prefix in $\mathbf{a}_{1\dots t}.a$, and $\mathrm{RNN}^{\mathrm{y}}, C^{\mathrm{y}}$ are similarly defined as in $\mathrm{RNN}^{\mathrm{x}}$ and C^{x}.

- X and Y are both strings — $X = \mathbf{x}, Y = \mathbf{y}$: in this case we let $C_{\boldsymbol{\phi}} = C^{\mathrm{xy}}(\mathbf{s}'_t, \mathrm{RNN}^{\mathrm{x}}(\mathbf{x}, i, \boldsymbol{\phi}), \mathrm{RNN}^{\mathrm{y}}(\mathbf{y}, j, \boldsymbol{\phi}); \boldsymbol{\phi})$.

Given a path prefix $\mathbf{a}_{:t-1}$, $q_t(a \mid \mathbf{a}_{:t-1}; \boldsymbol{\phi}, \mathcal{T}')$ is defined over arcs a such that $\mathbf{a}_{:t-1}.a$ is a valid path prefix in $\mathcal{T}'$. To optimize $\boldsymbol{\phi}$ with regard to $q_{\boldsymbol{\phi}}$, we follow (Lin and Eisner, 2018) and seek to find $\boldsymbol{\phi}^* = \mathrm{argmin}_{\boldsymbol{\phi}} \mathrm{KL}[\hat{p} || q_{\boldsymbol{\phi}}]$, where $\hat{p}$ is the approximate distribution defined in equation (7), which is equivalent to maximizing the log-likelihood of $q_{\boldsymbol{\phi}}(\mathbf{a})$ when $\mathbf{a}$ is distributed according to the approximation $\hat{p}$.

3.2 Training

In this paper, we consider joint training. The loss function of our model is defined as the negative log joint probability of string pair $(\mathbf{x}, \mathbf{y})$:

$$\mathcal{L}(\mathbf{x}, \mathbf{y}) = -\log p(\mathbf{x}, \mathbf{y}) = -\log \sum_{\mathbf{a} \in \mathbf{x} \circ \mathcal{T} \circ \mathbf{y}} p(\mathbf{a}). \tag{9}$$

Since p is an exponential family distribution, the gradients of $\mathcal{L}$ can be written as (Bishop, 2006)

$$\nabla \mathcal{L}(\mathbf{x}, \mathbf{y}) = -\mathbb{E}_{\mathbf{a} \sim p(\cdot \mid \mathbf{x} \circ \mathcal{T} \circ \mathbf{y})}[\nabla \log p(\mathbf{a})], \tag{10}$$

where $p(\cdot \mid \mathbf{x} \circ \mathcal{T} \circ \mathbf{y})$ is a conditioned distribution over paths. Computing equation (10) requires sampling from $p(\cdot \mid \mathbf{x} \circ \mathcal{T} \circ \mathbf{y})$, which, as we discuss in §3.1, is often impractical. We therefore approximate it with

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{x}, \mathbf{y}) &= -\mathbb{E}_{\mathbf{a} \sim p(\cdot \mid \mathbf{x} \circ \mathcal{T} \circ \mathbf{y})}[\nabla_{\boldsymbol{\theta}} \log p(\mathbf{a})] \\ &\approx -\mathbb{E}_{\mathbf{a} \sim \hat{p}(\cdot \mid \mathbf{x} \circ \mathcal{T} \circ \mathbf{y})}[\nabla_{\boldsymbol{\theta}} \log p(\mathbf{a})] \qquad (11) \\ &= -\sum_{m=1}^{M} w^{(m)} \nabla_{\boldsymbol{\theta}} G_{\boldsymbol{\theta}}(\mathbf{a}^{(m)}), \qquad (12)\end{aligned}$$

where q is a proposal distribution parametrized as in equation (8) (discussed in §3.1,) $\mathbf{a}^{(1)} \dots \mathbf{a}^{(M)} \sim q$ are i.i.d. samples of paths in $\mathbf{x} \circ \mathcal{T} \circ \mathbf{y}$, and $w^{(m)}$ is the **importance weight** of the m-th sample satisfying $w^{(m)} \propto \frac{\exp G_{\boldsymbol{\theta}}(\mathbf{a}^{(m)})}{q(\mathbf{a}^{(m)})}$, $\sum_{m=1}^{M} w^{(m)} = 1$. Pseudocode for calculating equation (12) is listed in Algorithm 1.

Algorithm 1 Compute approximate gradient for updating $G_{\boldsymbol{\theta}}$

Require: $G_{\boldsymbol{\theta}} : \mathcal{A} \to \mathbb{R}$ is an NFST scoring function, q is a distribution over paths, $M \in \mathbb{N}$ is the sample size

```
 1: function GET-GRADIENT($G_{\boldsymbol{\theta}}$, $M$, $q$)
 2:     for $m$ in $1 \dots M$ do
 3:         $\mathbf{a}^{(m)} \sim q$
 4:         $\tilde{w}^{(m)} \leftarrow \frac{\exp G_{\boldsymbol{\theta}}(\mathbf{a}^{(m)})}{q(\mathbf{a})}$
 5:     end for
 6:     $\hat{Z} \leftarrow \sum_{m=1}^{M} \tilde{w}^{(m)}$
 7:     for $m$ in $1 \dots M$ do
 8:         $w^{(m)} \leftarrow \frac{\tilde{w}^{(m)}}{\hat{Z}}$
 9:     end for
10:     return $-\sum_{m=1}^{M} w^{(m)} \nabla_{\boldsymbol{\theta}} G_{\boldsymbol{\theta}}(\mathbf{a}^{(m)})$
11: end function
```

3.3 Decoding most probable strings

Besides finding good *paths* in a conditioned distribution as we discuss in §3.1, we are also often interested in finding good *output strings*, which is conventionally referred to as the *decoding* problem, which we define to be finding the best output string $\mathbf{y}^* \triangleq \mathrm{argmax}_{\mathbf{y} \in L(Y)} p_{\mathbf{Y}}(\mathbf{y} \mid \mathcal{T}')$, where

$$p_{\mathbf{Y}}(\mathbf{y} \mid \mathcal{T}') \triangleq \frac{\sum_{\mathbf{a} \in \mathcal{T}' \circ \mathbf{y}} \tilde{p}(\mathbf{a})}{\sum_{\mathbf{a}' \in \mathcal{T}'} \tilde{p}(\mathbf{a}')}. \tag{13}$$

$\hat{\mathbf{y}}^* \triangleq \mathrm{argmax}_{\mathbf{y}} \hat{P}_{\mathbf{Y}}(\mathbf{y} \mid \mathcal{T}')$ is a consistent estimator of $\mathbf{y}^*$, which can directly be used to find the best string. However, making this estimate accurate might be expensive: it requires sampling many paths in the machine $\mathcal{T}'$, which is usually cyclic, and therefore has infinitely many more paths, than $\mathcal{T}' \circ \mathbf{y}_k$, which has finitely many paths when A is acyclic. On the other hand, for the task of finding the best string among a pool candidates, we do not need to compute (or approximate) the denominator in equation (13), since

$$\mathbf{y}^* = \underset{\mathbf{y} \in L(Y)}{\mathrm{argmax}} \sum_{\mathbf{a} \in \mathcal{T}' \circ \mathbf{y}} \tilde{p}(\mathbf{a}). \tag{14}$$

As in the case for paths, the language $L(Y)$ is usually infinitely large. However given an output candidate $\mathbf{y}_k \in L' \subseteq L(Y)$, we can approximate the summation in equation (14) using importance sampling:

$$\sum_{\mathbf{a} \in \mathcal{T}' \circ \mathbf{y}_k} \tilde{p}(\mathbf{a}) = \mathbb{E}_{\mathbf{a} \sim q(\cdot \mid \mathcal{T}' \circ \mathbf{y}_k)}[\frac{\tilde{p}(\mathbf{a})}{q(\mathbf{a} \mid \mathcal{T}' \circ \mathbf{y}_k)}], \tag{15}$$

Algorithm 2 Training procedure for $G_{\boldsymbol{\theta}}$. See Appendix C.2 for implementation details.

```
Require: $(\mathcal{T}, G_{\boldsymbol{\theta}})$ is an NFST, $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{y}_1) \ldots (\mathbf{x}_{|\mathcal{D}|}, \mathbf{y}_{|\mathcal{D}|})\}$ is the training dataset, LR : $\mathbb{N} \to \mathbb{R}$ is a
    learning rate scheduler, $\boldsymbol{\theta}_0$ are the initial parameters of $G_{\boldsymbol{\theta}}$, $M$ is a given sample size, maxEpoch $\in \mathbb{N}$
    is the number of epochs to train for
 1: procedure TRAIN($\mathcal{T}$, $G_{\boldsymbol{\theta}}$, $\mathcal{D}$, LR, $\boldsymbol{\theta}_0$, $M$, maxEpochs)
 2:     for epoch $\in [1 \ldots \mathrm{maxEpochs}]$ do
 3:         for $(\mathbf{x}_i, \mathbf{y}_i) \in \mathrm{shuffle}(\mathcal{D})$ do
 4:             $\mathcal{T}' \leftarrow \mathbf{x}_i \circ \mathcal{T} \circ \mathbf{y}_i$
 5:             Construct distribution $q(\cdot \mid \mathcal{T}')$ according to equation (8)
 6:             $\mathbf{u} \leftarrow$ GET-GRADIENT$(G_{\boldsymbol{\theta}}, M, q)$ (listed in Algorithm 1)
 7:             $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \mathrm{LR}(\mathrm{epoch}) \times \mathbf{u}$
 8:             (Optional) update the parameters of $q(\cdot \mid \mathcal{T}')$.
 9:         end for
10:     end for
11: end procedure
```

where $q(\cdot \mid \mathcal{T}' \circ \mathbf{y}_k)$ is a proposal distribution over paths in $\mathcal{T}' \circ \mathbf{y}_k$. In this paper we parametrize $q(\cdot \mid \mathcal{T}' \circ \mathbf{y}_k)$ following the definition in equation (8). When L' is finitely large, we reduce the decoding task into a reranking task.

To populate L', one possibility is to marginalize over paths in the approximate distribution $\hat{p}(\mathbf{a} \mid \mathcal{T}')$ discussed in §3.1 to obtain an estimate $\hat{p}_{\mathbf{Y}}(\mathbf{y} \mid \mathcal{T}')$, and use its support as L'. Note that it's possible to populate the candidate pool in other ways, each with its advantages and drawbacks: for example, one can use a top-k path set from a weighted (Markovian) FST. This approach guarantees exact computation, and the pool quality would no longer depend on the qualities of the smoothing distribution $q_{\boldsymbol{\phi}}$. However it is also a considerably much weaker model and may yield uninspiring candidates. In the common case where the conditioned machine $\mathcal{T}' = X \circ \mathcal{T} \circ Y$ has $X = \mathbf{x} \in \Sigma^*$ as the input string, and Y is the universal acceptor that accepts Δ^*, one can obtain a candidate pool from seq2seq models: seq2seq models can capture long distance dependencies between input and output strings, and are typically fast to train and decode from. However they are not applicable in the case where $L(Y) \neq \Delta^*$. Experimental details of decoding are further discussed in §4.3.

4 Experiments

Our experiments mainly aim to: (1) show the effectiveness of NFSTs on transduction tasks; (2) illustrate that how prior knowledge can be introduced into NFSTs and improve the performance; (3) demonstrate the interpretability of our model. Throughout, we experiment on three tasks: (i) grapheme-to-phoneme, (ii) phoneme-to-grapheme, and (iii) actions-to-commands. We compare with competitive string transduction baseline models in these tasks.

4.1 Tasks and datasets

We carry out experiments on three string transduction tasks:

Grapheme-to-phoneme and phoneme-to-grapheme (G2P/P2G) refer to the transduction between words' spelling and phonemic transcription. English has a highly irregular orthography (Venezky, 2011), which necessitates the use of rich models for this task. We use a portion of the standard CMUDict dataset: the Sphinx-compatible version of CMUDict (Weide, 2005). As for metrics, we choose widely used exact match accuracy and edit distance.

Action-to-command (A2C) refers to the transduction between an action sequence and imperative commands. We use NACS (Bastings et al., 2018) in our experiment. As for metrics, we use exact match accuracy (EM). Note that the in A2C setting, a given input can yield different outputs, e.g. `I_JUMP I_WALK I_WALK` corresponds to both "jump and walk twice" and "walk twice after jump". NACS is a finite set of action-command pairs; we consider a predicted command to be correct if it is in the finite set and its corresponding actions is exactly the input. We evaluate on the *length* setting proposed by Bastings et al. (2018), where we train on shorter sequences and evaluate on longer sequences.

4.2 FST designs

NFSTs require an unweighted FST $\mathcal{T}$ which defines a scaffold for the relation it recognizes. In this paper we experiment with two versions of $\mathcal{T}$: the first is a simple 'general' design $\mathcal{T}_0$, which contains only three states $s_{\{0,1,2\}}$, where the only arc between q_0 and q_1 consumes the mark <BOS>; and the only arc between q_1 and q_2 consumes the mark <EOS>. $\mathcal{T}_0$ has exactly one accepting state, which is q_2. To ensure that $\mathcal{T}_0$ defines relation for all possible string pairs $(\mathbf{x}, \mathbf{y}) \in \Sigma^* \times \Delta^*$, we add all arcs of the form $a = (s_1, s_1, \boldsymbol{\omega}, \sigma, \delta), \forall(\sigma, \delta) \in \Sigma \times \Delta$ to $\mathcal{T}$.

To recognize transduction rules defined in the Wikipedia English IPA Help page, we define $\mathcal{T}_{\text{IPA}}$, which has all states and arcs of $\mathcal{T}_0$, and additional states and arcs to handle multi-grapheme and multi-phoneme transductions defined in the IPA Help:[3] for example, the transduction th $\rightarrow$ θ is encoded as two arcs $(s_1, s_3, \boldsymbol{\omega}, \mathtt{t}, \theta)$ and $(s_3, s_1, \boldsymbol{\omega}, \mathtt{h}, \varepsilon)$. Because of the lack of good prior knowledge that can be added to A2C experiments, we only use general FSTs in those experiments for such experiments. Nor do we encode special marks that we are going to introduce below.[4]

4.2.1 Design of mark sequences

As with regular WFSTs, the arcs can often be hand-engineered to incorporate prior knowledge. Recall that as we describe in §2.2, each arc is associated with a mark sequence. In this paper, we will always derive the mark sequence on an arc $a = (s', s, \boldsymbol{\omega}', \sigma, \delta)$ of the transducer $\mathcal{T}$ as $\boldsymbol{\omega} = [\sigma, \boldsymbol{\omega}', \delta, s]$, where $\boldsymbol{\omega}' \in \Omega^*$ can be engineered to reflect FST- and application-specific properties of a path, such as the IPA Help list we mentioned earlier. One way to encode such knowledge into mark sequences is to have special mark symbols in mark sequences for particular transductions. In this paper we experiment with two schemes of marks:

- **IPA Help (IPA).** We define the IPA mark $\omega_{\text{IPA}} = \{\mathtt{C} \mid \mathtt{V}\}$, where the symbol C indicates that this arc is part of a transduction rule listed in the consonant section of the Wikipedia English IPA Help page. Similarly, the mark V indicates that the transduction rule is listed in the vowel section.

[3]https://en.wikipedia.org/wiki/Help:IPA/English

[4]The NACS dataset was actually generated from a regular transducer, which we could in principle use, but doing so would make the transduction fully deterministic and probably not interesting/hard enough.

- **Phoneme Classes (Phone).** We define Phone marks $\omega_{\text{Phone}} = \Phi(\delta)$, where Φ is a lookup function that returns the phoneme class of δ defined by the CMUDict dataset.[5]

In this paper we experiment with the following three FST and mark configurations for G2P/P2G experiments:

- **-IPA-Phone** in which case $\boldsymbol{\omega}' = \varnothing$ for all arcs. $\mathcal{T} = \mathcal{T}_0$.

- **+IPA-Phone** in which case $\boldsymbol{\omega}' = [\omega_{\text{IPA}}]$ when the transduction rule is found in the IPA Help list, otherwise $\boldsymbol{\omega}' = \varnothing$. $\mathcal{T} = \mathcal{T}_{\text{IPA}}$.

- **+IPA+Phone** in which case $\boldsymbol{\omega}' = [\omega_{\text{IPA}}\omega_{\text{Phone}}]$ when the transduction rule is found in the IPA Help list, otherwise $\boldsymbol{\omega}' = [\omega_{\text{Phone}}]$. $\mathcal{T} = \mathcal{T}_{\text{IPA}}$.

As we said earlier, we only use $\mathcal{T} = \mathcal{T}_0$ with no special marks for A2C experiments. Experimental results on these different configurations are in §5.3.

4.3 Decoding methods

We experiment with the following methods to decode the most probable strings:

- **Approximate Posterior (AP).** We approximate the posterior distribution over output strings $\hat{p}_{\mathbf{Y}}(\mathbf{y} \mid \mathcal{T}')$, and pick $\hat{\mathbf{y}}^* = \operatorname{argmax}_{\mathbf{y}} \hat{p}_{\mathbf{Y}}(\mathbf{y} \mid \mathcal{T}')$ as the output.

- **Reranking AP.** As we discuss in §3.3, improving $\hat{\mathbf{y}}^*$ by taking more path samples in $\mathcal{T}'$ may be expensive. The reranking method uses the support of $\hat{p}_{\mathbf{Y}}$ as a candidate pool L', and for each $\mathbf{y}_k \in L'$ we estimate equation (15) using path samples in $\mathcal{T}' \circ \mathbf{y}_k$.

- **Reranking External.** This decoding method uses k-best lists from external models. In this paper, we make use of sequence-to-sequence baseline models as the candidate pool L'.

- **Reranking AP + External.** This decoding method uses the union of the support of $\hat{p}_{\mathbf{Y}}$ and k-best lists from the sequence-to-sequence baseline models as the candidate pool L'.

In this paper, we take 128 path samples per candidate for all **Reranking** methods.

[5]https://github.com/cmusphinx/cmudict/blob/master/cmudict.phones

5 Results

5.1 Baselines

We compare NFSTs against the following baselines:

BiRNN-WFSTs proposed by Rastogi et al. (2016), were weighted finite-state transducers whose weights encode input string features by the use of recurrent neural networks. As we note in Table 1, they can be seen as a special case of NFSTs, where the Markov property is kept, but where exact inference is still possible.

Seq2seq models are the standard toolkit for transduction tasks. We make use of the attention mechanism proposed by Luong et al. (2015), which accomplishes 'soft alignments' that do not enforce a monotonic alignment constraint.

Neuralized IBM Model 1 is a character transduction model recently proposed by Wu et al. (2018), which marginalizes over non-monotonic hard alignments between input and output strings. Like (Luong et al., 2015), they did not enforce monotonic alignment constraints; but unlike them, they did not make use of the **input feeding** mechanism,[6] where past alignment information is fed back into the RNN decoder. This particular omission allows (Wu et al., 2018) to do exact inference with a dynamic programming algorithm.

All baseline systems are tuned on the validation sets. The seq2seq models employ GRUs, with word and RNN embedding size = 500 and a dropout rate of 0.3. They are trained with the Adam optimizer (Kingma and Ba, 2014) over 50 epochs. The Neuralized IBM Model 1 models are tuned as described in (Wu et al., 2018).

5.2 The effectiveness of NFSTs

5.2.1 Does losing the Markov property help?

Table 2 indicates that BiRNN-WFST models (Rastogi et al., 2016) perform worse than other models. Their Markovian assumption helps enable dynamic programming, but restricts their expressive power, which greatly hampers the BiRNN-WFST's performance on the P2G/G2P task. The NACS task also relies highly on output-output interactions, and BiRNN-WFST performs very poorly there.

[6]We discuss this further in Appendix B.1.

	G2P / P2G				NACS
	EM Accuracy		Edit Distance		EM Accuracy
	Dev	Test	Dev	Test	Test
BiRNN-WFST	16.9	15.9	1.532	1.645	5.6
Seq2seq	30.7	28.9	1.373	1.426	9.0
Neuralized IBM Model 1	31.6	30.2	1.366	1.398	—
Local NFSTs	32.7	31.8	1.319	1.332	15.64

Table 2: Average exact match accuracy (%, higher the better) and edit distance (lower the better) on G2P and P2G as well as exact match accuracy on NACS. Comparison between our models with baselines. For NFST models, we make use of the **Reranking AP** decoding method described in §4.2.

5.2.2 Effectiveness of proposed decoding methods

Table 3 shows results from different decoding methods on the G2P/P2G tasks, configuration **+IPA+PHONE**. **AP** performs significantly worse than **Reranking AP**, suggesting that the estimate $\hat{\mathbf{y}}^*$ suffers from the variance problem. Interestingly, of decoding methods that employ external models, **Reranking External** performs better than **Reranking AP + External**, despite having a smaller candidate pool. We think there is some product-of-experts effect in **Reranking External** since the external model may not be biased in the same way as our model is. But such benefits vanish when candidates from **AP** are also in the pool — our learned approximation learns the bias in the model — and hence the worse performance in **Reranking AP + External**. This suggests an interesting regularization trick in practice: populating the candidate pool using external models to hide our model bias. However when we compare our method against non-NFST baseline methods we do not make use of such tricks, to ensure a more fair comparison.

	EM Accuracy		Edit Distance	
	Dev	Test	Dev	Test
AP	28.2	28.2	1.513	1.467
Reranking AP	32.7	31.8	1.319	1.332
Reranking External	33.3	32.7	1.297	1.298
Reranking AP + External	32.9	32.0	1.309	1.303

Table 3: Average exact match accuracy (%, higher the better) and edit distance (lower the better) on G2P and P2G. The effectiveness of different decoding methods.

5.3 Prior knowledge: does it help?

In Table 4 we see that combining both +IPA and +PHONE improves model generalizability over the general FST (-IPA -PHONE). We also note that using only the IPA marks leads to degraded performance

	EM Accuracy		Edit Distance	
	Dev	Test	Dev	Test
-IPA -PHONE	31.8	29.3	1.38	1.373
+IPA -PHONE	31.3	29.2	1.367	1.431
+IPA +PHONE	32.7	31.8	1.319	1.332

Table 4: Average exact match accuracy (%, higher the better) and edit distance (lower the better) on G2P and P2G. The effectiveness of different FST designs.

compared to the general FST baseline. This is a surprising result — one explanation is the IPA marks are not defined on all paths that transduce the intended input-output pairs: NFSTs are capable of recognizing phoneme-grapheme alignments in different paths,[7] but only one such path is marked by +IPA. But we leave a more thorough analysis to future work.

6 Related Work

Recently, there has been work relating finite-state methods and neural architectures. For example, Schwartz et al. (2018) and Peng et al. (2018) have shown the equivalence between some neural models and WFSAs. The most important differences of our work is that in addition to classifying strings, NFSTs can also transduce strings. Moreover, NFSTs also allow free topology of FST design, and breaks the Markovian assumption. In addition to models we compare against in §4, we note that (Aharoni and Goldberg, 2017; Deng et al., 2018) are also similar to our work; in that they also marginalize over latent alignments, although they do not enforce the monotonicity constraint. Work that discusses globally normalized sequence models are relevant to our work. In this paper, we discuss a training strategy that bounds the partition function; other ways to train a globally normalized model (not necessarily probabilistic) include (Wiseman and Rush, 2016; Andor et al., 2016). On the other hand, our locally normalized FSTs bear resemblance to (Dyer et al., 2016), which was also locally normalized, and also employed importance sampling for training.

7 Conclusions and Future Work

Neural finite state transducers (NFSTs) are able to model string pairs, considering their monotonic alignment but also enjoying RNNs' power to handle non-finite-state phenomena. They compete favorably with state-of-the-art neural models on transduction tasks. At the same time, it is easy to inject domain knowledge into NFSTs for inductive bias, and they offer interpretable paths.

In this paper, we have used rather simple architectures for our RNNs; one could experiment with multiple layers and attention. One could also experiment with associating marks differently with arcs—the marks are able to convey useful domain information to the RNNs. For example, in a P2G or G2P task, all arcs that cross a syllable boundary might update the RNN state using a `syllable` mark. We envision using regular expressions to build the NFSTs, and embedding marks in the regular expressions as a way of sending useful features to the RNNs to help them evaluate paths.

In this paper, we have studied NFSTs as standalone systems. But as probabilistic models, they can be readily embedded in a bigger picture: it should be directly feasible to incorporate a globally/locally normalized NFST in a larger probabilistic model (Finkel and Manning, 2009; Chiang et al., 2010).

The path weights of NFSTs could be interpreted simply as scores, rather than log-probabilities. One would then decode by seeking the 1-best path with input $\mathbf{x}$, e.g., via beam search or Monte Carlo Tree Search. In this setting, one might attempt to train the NFST using methods similar to the max-violation structured perceptron or the structured SVM.

Acknowledgments

This work has been generously supported by a Google Faculty Research Award and by Grant No. 1718846 from the National Science Foundation, both to the last author. Hao Zhu is supported by Tsinghua University Initiative Scientific Research Program. We thank Shijie Wu for providing us IBM Neuralized Model 1 experiment results.

References

Roee Aharoni and Yoav Goldberg. 2017. Morphological inflection generation with hard monotonic attention. In *ACL*.

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally normalized transition-based neural networks. In *Association for Computational Linguistics*.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of ICLR*.

[7]This is discussed further in Appendix B.2.

Joost Bastings, Marco Baroni, Jason Weston, Kyunghyun Cho, and Douwe Kiela. 2018. Jump to better conclusions: Scan both left and right. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 47–55.

Jean Berstel, Jr. and Christophe Reutenauer. 1988. *Rational Series and Their Languages*. Springer-Verlag, Berlin, Heidelberg.

Christopher M Bishop. 2006. Pattern recognition and machine learning.

David Chiang, Jonathan Graehl, Kevin Knight, Adam Pauls, and Sujith Ravi. 2010. Bayesian inference for finite-state transducers. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 447–455. Association for Computational Linguistics.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *EMNLP*, pages 1724–1734. ACL.

Ryan Cotterell, Nanyun Peng, and Jason Eisner. 2014. Stochastic contextual edit distance and probabilistic FSTs. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 625–630, Baltimore.

Yuntian Deng, Yoon Kim, Justin Chiu, Demi Guo, and Alexander Rush. 2018. Latent alignment and variational attention. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 9735–9747. Curran Associates, Inc.

Markus Dreyer. 2011. *A Non-Parametric Model for the Discovery of Inflectional Paradigms from Plain Text Using Graphical Models over Strings*. Ph.D. thesis, Johns Hopkins University, Baltimore, MD.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. 2016. Recurrent neural network grammars. In *HLT-NAACL*.

Jenny Rose Finkel and Christopher D Manning. 2009. Hierarchical bayesian domain adaptation. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 602–610. Association for Computational Linguistics.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Chu-Cheng Lin and Jason Eisner. 2018. Neural particle smoothing for sampling from conditional sequence models. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 929–941, New Orleans.

Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421.

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

Hao Peng, Roy Schwartz, Sam Thomson, and Noah A. Smith. 2018. Rational recurrences. In *EMNLP*.

Pushpendre Rastogi, Ryan Cotterell, and Jason Eisner. 2016. Weighting finite-state transductions with neural context. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 623–633, San Diego. 11 pages. Supplementary material (1 page) also available.

Emmanuel Roche and Yves Schabes, editors. 1997. *Finite-State Language Processing*. MIT Press.

Roy Schwartz, Sam Thomson, and Noah A. Smith. 2018. Sopa: Bridging cnns, rnns, and weighted finite-state machines. *CoRR*, abs/1805.06061.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 3104–3112, Cambridge, MA, USA. MIT Press.

Ottokar Tilk and Tanel Alumäe. 2016. Bidirectional recurrent neural network with attention mechanism for punctuation restoration. In *INTERSPEECH*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need.

Richard L Venezky. 2011. *The structure of English orthography*, volume 82. Walter de Gruyter.

Robert Weide. 2005. The carnegie mellon pronouncing dictionary [cmudict. 0.6].

Sam Wiseman and Alexander M. Rush. 2016. Sequence-to-sequence learning as beam-search optimization. In *EMNLP*.

Shijie Wu, Pamela Shapiro, and Ryan Cotterell. 2018. Hard non-monotonic attention for character-level transduction. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4425–4438.

A Finite-state transducers

A.1 Rational Relations

A **relation** is a set of pairs—in this paper, a subset of $\Sigma^* \times \Delta^*$, so it relates strings over an "input" alphabet Σ to strings over an "output" alphabet Δ.

A **weighted relation** is a function R that maps any string pair $(\mathbf{x}, \mathbf{y})$ to a **weight** in $\mathbb{R}_{\geq 0}$.

We say that the relation R is **rational** if R can be defined by some weighted finite-state transducer (FST) $\mathcal{T}$. As formalized in Appendix A.3, this means that $R(\mathbf{x}, \mathbf{y})$ is the total weight of all accepting paths in $\mathcal{T}$ that are labeled with $(\mathbf{x}, \mathbf{y})$ (which is 0 if there are no such accepting paths). The weight of each accepting path in $\mathcal{T}$ is given by the product of its arc weights, which fall in $\mathbb{R}_{>0}$.

The set of pairs $\text{support}(R) \triangleq \{(\mathbf{x}, \mathbf{y}) : R(\mathbf{x}, \mathbf{y}) > 0\}$ is then said to be a **regular relation** because it is recognized by the unweighted FST obtained by dropping the weights from $\mathcal{T}$. In this paper, we are interested in defining *non-rational* weighting functions R with this same regular support set.

A.2 Finite-state transducers

We briefly review finite-state transducers (FSTs). Formally, an FST is a tuple $\mathcal{T}_0 = (\Sigma, \Delta, Q, A_0, I, F)$ where

- Σ is a finite input alphabet
- Δ is a finite output alphabet
- Q is a finite set of states
- $A_0 \subseteq Q \times Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\})$ is the set of weighted arcs
- $I \subseteq Q$ is the set of initial states (conventionally $|I| = 1$)
- $F \subseteq Q$ is the set of final states

Let $\mathbf{a} = a_1 \ldots a_T$ (for $T \geq 0$) be an accepting path in $\mathcal{T}_0$, that is, each $a_i = (q_{i-1}, q_i, \sigma_i, \delta_i) \in A_0$ and $q_0 \in I, q_T \in F$. We say that the input and output strings of $\mathbf{a}$ are $\sigma_1 \cdots \sigma_T$ and $\delta_1 \cdots \delta_T$.

A.3 Real-valued weighted FSTs

Weighted FSTs (WFSTs) are defined very similarly to FSTs. A WFST is formally defined as a 6-tuple, just like an (unweighted) FST: $\mathcal{T} = (\Sigma, \Delta, Q, A, I, F)$, with arcs carrying weights: $A \subseteq Q \times Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\}) \times \mathbb{R}$. Compared to FST arcs in Appendix A.2, a WFST arc each $a_i = (q_{i-1}, q_i, \sigma_i, \delta_i, \kappa_i) \in A$ has weight κ_i. We also define the weight of $\mathbf{a}$ to be $w(\mathbf{a}) \triangleq \bigotimes_{i=1}^{T} \kappa_i \in \mathbb{R}$.

The weight of the entire WFST $\mathcal{T}$ is defined as the total weight (under $\oplus$) of all accepting paths:

$$\mathcal{T}[\,] \triangleq \bigoplus_{\mathbf{a}} w(\mathbf{a}) \in \mathbb{R} \qquad (16)$$

More interestingly, the weight $\mathcal{T}[\mathbf{x}, \mathbf{y}]$ of a string pair $\mathbf{x} \in \Sigma^*, \mathbf{y} \in \Delta^*$ is given by similarly summing $w(\mathbf{a})$ over just the accepting paths $\mathbf{a}$ whose input string is $\mathbf{x}$ and output string is $\mathbf{y}$.

B More analysis on the effectiveness of NFSTs

B.1 Does feeding alignments into the decoder help?

In particular, we attribute our models' outperforming Neuralized IBM Model 1 to the fact that a complete history of past alignments is remembered in the RNN state. (Wu et al., 2018) noted that in character transduction tasks, past alignment information seemed to barely affect decoding decisions made afterwards. However, we empirically find that there is performance gain by explicitly modeling past alignments. This also shows up in our preliminary experiments with non-input-feeding seq2seq models, which resulted in about 1% of lowered accuracy and about 0.1 longer edit distance.

B.2 Interpretability of learned paths

The model is not required to learn transduction rules that conform to our linguistic knowledge. However, we expect that a well-performing one would tend to pick up rules that resemble what we know. To verify this, we obtain samples (listed in Table 4) from $\hat{p}(\mathbf{a} \mid \mathbf{x}, \mathbf{y})$ using the importance sampling algorithm described in §3.3. We find that our NFST model has learned to align phonemes and graphemes, generating them alternately. It has no problem picking up obvious pairs in the English orthography (e.g. (ʃ, `c h`), and (ŋ, `n g`)). We also find evidence that the model has picked up how context affects alignment: for example, the model has learned that the bigram 'gh' is pronounced differently in different contexts: in 'onslau**ght**,' it is aligned with ɔ in the sequence 'augh;' in 'Willin**gh**am,' it spans over two phonemes ŋ h; and in '**gh**ezzi,' it is aligned with the phoneme g. We also find that our NFST has no problem learning phoneme-grapheme alignments that span over two

Input / Output	Paths	$\hat{P}(\mathbf{a} \mid \mathbf{x}, \mathbf{y})$
/mɑɹʃ/ marche	ϵ:m m:ɑ a:ɹ r:ʃ c:ϵ h:ϵ e:ϵ	96.5%
	ϵ:m m:ɑ a:ɹ r:ϵ ϵ:ʃ c:ϵ h:ϵ e:ϵ	2.5%
	ϵ:m m:ɑ a:ϵ ϵ:ɹ r:ʃ c:ϵ h:ϵ e:ϵ	1.0%
/ɔnslɔt/ onslaught	ϵ:ɔ o:n n:ϵ ϵ:s s:l l:ɔ a:ϵ u:ϵ g:ϵ h:t t:ϵ	76.3%
	ϵ:ɔ o:n n:s s:l l:ɔ a:ϵ u:ϵ g:ϵ h:t t:ϵ	21.4%
	ϵ:ɔ o:n n:ϵ ϵ:s s:l l:ɔ a:ϵ u:ϵ g:ϵ h:ϵ ϵ:t t:ϵ	1.5%
/wɪlɪŋhəm/ Willingham	ϵ:w W:ɪ i:l l:ϵ l:ϵ ϵ:ɪ i:ŋ n:ϵ g:ϵ ϵ:h h:ə a:ϵ ϵ:m m:ϵ	40.1%
	ϵ:w W:ɪ i:l l:ϵ l:ɪ i:ŋ n:ϵ g:ϵ ϵ:h h:ə a:ϵ ϵ:m m:ϵ	36.6%
	ϵ:w W:ɪ i:l l:ϵ l:ɪ i:ŋ n:ϵ g:h h:ə a:ϵ ϵ:m m:ϵ	7.4%
/gezɪ/ ghezzi	ϵ:g g:ϵ h:e e:z z:ϵ ɪ:z i:ϵ	98.8%
	ϵ:g g:e h:ϵ e:z z:ɪ z:ϵ i:ϵ	1.2%

Table 5: Most probable paths from $\mathbf{x} \circ \mathcal{T} \circ \mathbf{y}$ under the approximate posterior distribution.

arcs, which is beyond the capability of of ordinary WFSTs.

C Implementation Details

C.1 Model parametrization details

As mentioned before, the type of RNN that we use is GRU. The GRU parameterizing $G_{\boldsymbol{\theta}}$ has 500 hidden states. The embedding sizes of tokens, including the input symbol, output symbol and states, and marks are all 500. During inference we make use of proposal distributions $q_{\boldsymbol{\phi}}(\mathbf{a} \mid \mathcal{T}')$, where $\mathcal{T}' \in \{\mathbf{x} \circ \mathcal{T}, \mathcal{T} \circ \mathbf{y}, \mathbf{x} \circ \mathcal{T} \circ \mathbf{y}\}$. All RNNs used to parametrize $q_{\boldsymbol{\phi}}$ are also GRUs, with 125 hidden states. $q_{\boldsymbol{\phi}}$ makes use of input/output embeddings independent from $G_{\boldsymbol{\theta}}$, which also have size 125 in this paper. The feed-forward networks $C^{\mathrm{x,y,xy}}$ are parametrized by 3-layer networks, with ReLU as the activation function of the first two layers. The output dimension sizes of the first and second layers are $\lfloor D/2 \rfloor$ and $\lfloor D/4 \rfloor$, where D is the input vector dimension size.

C.2 Training procedure details

We use stochastic gradient descent (SGD) to train $G_{\boldsymbol{\theta}}$. For each example, we compute the gradient using normalized importance sampling over an ensemble of 512 particles (paths), the maximum that we could compute in parallel. By using a large ensemble, we reduce both the bias (from normalized importance sampling) and the variance of the gradient estimate; we found that smaller ensembles did not work as well. Thus, we used only one example per minibatch.

We train the 'clamped' proposal distribution $q_{\boldsymbol{\phi}}(\mathbf{a} \mid \mathbf{x} \circ \mathcal{T} \circ \mathbf{y})$ differently from the 'free' ones $q_{\boldsymbol{\phi}}(\mathbf{a} \mid \mathbf{x} \circ \mathcal{T})$ and $q_{\boldsymbol{\phi}}(\mathbf{a} \mid \mathcal{T} \circ \mathbf{y})$. The clamped distribution is trained alternately with $G_{\boldsymbol{\theta}}$, as listed in Algorithm 2. We evaluate on the development dataset at the end of each epoch using the **Reranking External** method described in §4.3. When the EM accuracy stops improving, we fix the parameters of $G_{\boldsymbol{\theta}}$ and start training $q_{\boldsymbol{\phi}}(\mathbf{x} \circ \mathcal{T})$ and $q_{\boldsymbol{\phi}}(\mathcal{T} \circ \mathbf{y})$ on the inclusive KL divergence objective function, using methods described in (Lin and Eisner, 2018). We then initialize the free distributions' RNNs using those of the clamped distributions. We train the free proposal distributions for 30 epochs, and evaluate on the development dataset at the end of each epoch. Results from the best epochs are reported in this paper.