Robustness to Programmable String Transformations via Augmented Abstract Training

Yuhao Zhang 1 Aws Albarghouthi 1 Loris D’Antoni 1

Abstract
Deep neural networks for natural language processing tasks are vulnerable to adversarial input perturbations. In this paper, we present a versatile language for programmatically specifying string transformations—e.g., insertions, deletions, substitutions, swaps, etc.—that are relevant to the task at hand. We then present an approach to adversarially training models that are robust to such user-defined string transformations. Our approach combines the advantages of search-based techniques for adversarial training with abstraction-based techniques. Specifically, we show how to decompose a set of user-defined string transformations into two component specifications, one that benefits from search and another from abstraction. We use our technique to train models on the AG and SST2 datasets and show that the resulting models are robust to combinations of user-defined transformations mimicking spelling mistakes and other meaning-preserving transformations.

1. Introduction
Deep neural networks have proven incredibly powerful in a huge range of machine-learning tasks. However, deep neural networks are highly sensitive to small input perturbations that cause the network’s accuracy to plummet (Carlini & Wagner, 2017; Szegedy et al., 2013). In the context of natural language processing, these adversarial examples come in the form of spelling mistakes, use of synonyms, etc.—essentially, meaning-preserving transformations that cause the network to change its prediction (Ebrahimi et al., 2018; Zhang et al., 2019a; Michel et al., 2019).

The question we ask in this paper is:

Can we train models that are robust against rich perturbation spaces over strings?

The practical challenge in answering this question is computing the worst-case loss. This is because the perturbation space $R(x)$ can be enormous and therefore impractical to enumerate. This is particularly true for NLP tasks, where the perturbation space $R(x)$ should contain inputs that are semantically equivalent to $x$—e.g., variations of the sentence $x$ with typos or words replaced by synonyms. Therefore, we need to approximate the adversarial loss. There are two such classes of approximation techniques:

Augmentation The first class of techniques computes a lower bound on the adversarial loss by exploring a finite number of points in $R(x)$. This is usually done by applying a gradient-based attack, like HotFlip (Ebrahimi et al., 2018) for natural-language tasks or PGD (Madry et al., 2018) for computer-vision tasks. We call this class of techniques augmentation-based, as they essentially search for a perturbed sample with which to augment the training set.

Abstraction The second class of techniques computes an...
Robustness to String Transformations via Augmented Abstract Training

![Diagram showing perturbation space, augmentation, abstraction, and A3T]

Figure 1. Illustration of augmentation, abstraction, and A3T

*upper bound* on the adversarial loss by overapproximating, or *abstracting*, the perturbation space $R(x)$ into a set of symbolic constraints that can be efficiently propagated through the network. For example, the *interval* abstraction has been used in numerous works (Mirman et al., 2018; Gowal et al., 2019; Huang et al., 2019). We call this class of techniques *abstraction*-based.

Both classes of techniques can produce suboptimal results: augmentation can severely underapproximate the worst-case loss and abstraction can severely overapproximate the loss. Particularly, we observe that the two techniques have complementary utility, working well on some perturbation spaces but not others—for example, Huang et al. (2019) have shown that abstraction works better for token substitutions, while augmentation-based techniques like HotFlip (Ebrahimi et al., 2018) and MHA (Zhang et al., 2019b) are general—e.g., apply to token deletions and insertions.

1.1. Our Approach

A hybrid approach We propose augmented abstract adversarial training (A3T), an adversarial training technique that combines the strengths of augmentation and abstraction techniques. The key idea underlying A3T is to decompose the perturbation space into two subsets, one that can be explored using augmentation and one that can be abstracted—e.g., using augmentation to explore word duplication typos and abstraction to explore replacing words with synonyms. From an algorithmic perspective, our computation of adversarial loss switches from a concrete, e.g., gradient-based, search through the perturbation space to a symbolic search. As such, for every training sample $(x, y)$, our technique may end up with a lower bound or an upper bound on its adversarial loss (see Fig. 1).

A language for specifying string transformations The challenge of applying A3T is how to exactly decompose the perturbation space. Our key enabling idea is to define the perturbation space *programmatically*, in a way that can be easily and cleanly decomposed. Specifically, we define a general language in which we can specify a perturbation space by a specification $S$ in the form of $\{T_1, \delta_1\}, \ldots, \{T_n, \delta_n\}$, containing a set of string transformations $T_i : \mathcal{X} \rightarrow 2^\mathcal{X}$. The specification $S$ defines a perturbation space of all possible strings by applying each transformation $T_i$ up to $\delta_i$ times. For example, given a string $x$, $S(x)$ could define the set of all strings $x'$ that are like $x$ but with some words replaced by one of its synonyms and with some stop words removed.

Given a perturbation space defined by a set of transformations, A3T decomposes the set of transformations into two disjoint subsets, one that is explored concretely (augmentation) and one that is explored symbolically (abstraction).

**Results** We have implemented A3T and used it to train NLP models for sentiment analysis that are robust to a range of string transformations—e.g., character swaps modeling spelling mistakes, substituting of a word with a synonym, removing stop words, duplicating words, etc. Our results show that A3T can train models that are more robust to adversarial string transformations than those produced using existing techniques.

1.2. Summary of Contributions

- We present A3T, a technique for training models that are robust to string transformations. A3T combines search-based attacks and abstraction-based techniques to explore the perturbation space and compute good approximations of adversarial loss.
- To enable A3T, we define a general language of string transformations with which we can specify the perturbation space. A3T exploits the specification to decompose and search the perturbation space.
- We implement A3T\(^1\) and evaluate it on two datasets and a variety of string transformations. Our results demonstrate the increase in robustness achieved by A3T in comparison with state-of-the-art techniques.

2. Related Work

Adversarial text generation Zhang et al. (2019b) presented a comprehensive overview of adversarial attacks on neural networks over natural language. In this paper, we focus on the word- and character-level. HotFlip (Ebrahimi et al., 2018) is a gradient-based approach that can generate the adversarial text in the perturbation space described by word- and character-level transformations. MHA (Zhang et al., 2019a) uses Metropolis-Hastings sampling guided by gradients to generate word-level adversarial text via word substitution. Karpukhin et al. (2019) designed character-

\(^1\)We provide our code at [https://github.com/Forev er2yh/A3T](https://github.com/Forever2yh/A3T).
level work focuses on generating adversarial text on the sentence-level (Liang et al., 2018) or paraphrase-level (Iyyer et al., 2018; Ribeiro et al., 2018). Also, some works (Zhao et al., 2018; Wang et al., 2019) try to generate natural or grammatically correct adversarial text.

**Abstract training** Mirman et al. (2018) and Gowal et al. (2020) proposed the formal verification under text deletion (IBP) to train image classification models that are provably \( \theta \) is to find the optimal parameters \( Y \). We consider a standard classification setting and (2) define a string transformation language for specifying the perturbation space.

**3. The Perturbation-Robustness Problem**

In this section, we (1) formalize the perturbation-robustness problem and (2) define a string transformation language for specifying the perturbation space.

**3.1. Perturbation Robustness**

**Classification setting** We consider a standard classification setting with samples from some domain \( \mathcal{X} \) and labels from \( \mathcal{Y} \). Given a distribution \( D \) over samples and labels, our goal is to find the optimal parameters \( \theta \) of some neural-network architecture \( F_\theta \) that minimize the expected loss

\[
\argmin_{\theta} \mathbb{E}_{(x,y) \sim D} \mathcal{L}(x, y, \theta) \quad (1)
\]

We are interested in the setting where the sample space \( \mathcal{X} \) defines strings over some finite alphabet \( \Sigma \). The alphabet \( \Sigma \) can be, for example, English characters (in a character-level model) or entire words (in a world-level model). Therefore, the domain \( \mathcal{X} \) in our setting is \( \Sigma^n \), i.e., the set of all strings of elements of \( \Sigma \). We will use \( x \in \Sigma^n \) to denote a string and \( x_i \in \Sigma \) to denote the \( i \)th element of the string.

**Perturbation space** We define a perturbation space \( R \) as a function in \( \Sigma^n \to 2^\Sigma \), i.e., \( R \) takes a string \( x \) and returns a finite set of possible perturbed strings of \( x \).

We will use a perturbation space to denote a set of strings that should receive the same classification by our network. For example, \( R(x) \) could define a set of sentences paraphrasing \( x \). We can thus modify our training objective into a robust-optimization problem, following Madry et al. (2018):

\[
\argmin_{\theta} \mathbb{E}_{(x,y) \sim D} \max_{z \in R(x)} \mathcal{L}(z, y, \theta) \quad (2)
\]

This inner objective is usually hard to solve; in our setting, the perturbation space can be very large and we cannot afford to consider every single point in that space during training. Therefore, as we discussed in Section 1, typically approximations are made.

**Exhaustive accuracy** Once we have trained a model \( F_\theta \) using the robust optimization objective, we will use exhaustive accuracy to quantify its classification accuracy in the face of perturbations. Specifically, given a dataset \( D = \{(x_i, y_i)\}_{i=1}^n \) and a perturbation space \( R \), we define exhaustive accuracy as follows:

\[
\frac{1}{n} \sum_{i=1}^n 1 \left[ \forall z \in R(x_i), F_\theta(z) = y_i \right] \quad (3)
\]

Intuitively, for each sample \( (x_i, y_i) \), its classification is considered correct if \( F_\theta \) predicts \( y_i \) for every single point in \( R(x_i) \). We use exhaustive accuracy instead of the commonly used adversarial accuracy because (1) exhaustive accuracy provides the ground truth accuracy of the discrete perturbation spaces and does not depend on an underlying adversarial attack, and (2) the discrete spaces make it easy for us to compute exhaustive accuracy by enumeration and at the same time hard for the gradient-based adversarial attacks to explore the space.

**3.2. A Language for Specifying Perturbations**

We have thus far assumed that the perturbation space is provided to us. We now describe a language for modularly specifying a perturbation space.
A specification $S$ is defined as follows:

$$S = \{(T_1, \delta_1), \ldots, (T_n, \delta_n)\}$$

where each $T_i$ denotes a string transformation that can be applied up to $\delta_i \in \mathbb{N}$ times. Formally, a string transformation $T$ is a pair $(\varphi, f)$, where $\varphi : \Sigma^* \rightarrow \{0, 1\}$ is a Boolean predicate (in practice, a regular expression) describing the substrings of the inputs to which the transformation can be applied, and $f : \Sigma^* \rightarrow 2^{\Sigma^*}$ is a transformer describing how the substrings matched by $\varphi$ can be replaced.

**Single transformations** Before defining the semantics of our specification language, we illustrate a few example specifications involving single transformations:

**Example 1** ($T_{\text{stop}} = (\varphi_{\text{stop}}, f_{\text{stop}})$). Suppose we want to define a transformation that deletes a stop word—and, the, is, etc.—mimicking a typo. The predicate $\varphi_{\text{stop}}$ will be a regular expression matching all stop words. The transformer $f_{\text{stop}}$ will be simply the function that takes a string and returns the set containing the empty string, $f_{\text{stop}}(x) = \{\epsilon\}$. Consider a specification $S_{\text{stop}} = \{(T_{\text{stop}}, 1)\}$ that applies the transformation $T_{\text{stop}}$ up to one time. On the following string, *They are at school*, the predicate $\varphi_{\text{stop}}$ matches the substrings *are* and *at*. In both cases, we apply the predicate $f_{\text{stop}}$ to the matched word and insert the output of $f_{\text{stop}}$ in its position. This results in the set containing the original string (0 transformations are applied) and the two strings *They at school* and *They are school*. Applying a specification $S_{\text{stop}}^2 = \{(T_{\text{stop}}, 2)\}$, which is allowed to apply $T_{\text{stop}}$ at most twice, to the same input would result in a set of strings containing the strings above as well as the string *They school*.

**Example 2** ($T_{\text{nice}} = (\varphi_{\text{nice}}, f_{\text{nice}})$). Say we want to transform occurrences of *nice* into one of its synonyms, enjoyable and pleasant. We define the predicate $\varphi_{\text{nice}}(x)$ that is true iff $x = \text{n}i\text{c}e$, and we define $f_{\text{nice}}(x) = \{\text{enjoyable}, \text{pleasant}\}$. Given the string *This is nice!*, it will be transformed into the set *This is enjoyable! This is pleasant!*.

Applying a specification $S_{\text{nice}}^2 = \{(T_{\text{nice}}, 2)\}$ to the same input would result in the same set of strings above. Because the predicate $\varphi_{\text{nice}}(x)$ only matches the word *nice*.

**Example 3** ($T_{\text{swap}} = (\varphi_{\text{swap}}, f_{\text{swap}})$). Now consider the case where we would like to swap adjacent vowels. $\varphi_{\text{swap}}$ will be defined as the regular expression that matches two adjacent characters that are vowels. Next, since $x = x_0x_1$ can only have length 2, the transformer $f_{\text{swap}}$ will be the swap function $f_{\text{swap}}(x_0x_1) = \{x_1x_0\}$.

**Multiple transformations** As discussed above, a specification $S$ in our language is a set of transformations $\{(T_1, \delta_1), \ldots, (T_n, \delta_n)\}$ where each $T_i$ is a pair $(\varphi_i, f_i)$. The formal semantics of our language can be found in the supplementary Appendix. Informally, a string $z$ is in the perturbation space $S(x)$ if it can be obtained by (1) finding a set $\sigma$ of non-overlapping substrings of $x$ that match the various predicates $\varphi_i$ and such that at most $\delta_i$ substrings in $\sigma$ are matches of $\varphi_i$, and (2) replacing each substring $x' \in \sigma$ matched by $\varphi_i$ with a string in $f_i(x')$. The complexity of the formalization is due to the requirement that matched substrings should not overlap—this requirement guarantees that each character in the input is only involved in a single transformation and will be useful when formalizing our abstract training approach in Section 4.2.

**Example 4 (Multiple Transformations).** Using the transformations $T_{\text{nice}}$ and $T_{\text{swap}}$ we can define the specification

$$S_{\text{ns}} = \{(T_{\text{nice}}, 1), (T_{\text{swap}}, 1)\}$$

Then, $S_{\text{ns}}(\text{This house is nice})$ results in the set of strings:

- **This house is nice**
- **This house is enjoyable**
- **This house is pleasant**
- **This house is nice**
- **This house is enjoyable**
- **This house is pleasant**

The transformed portions are shown in bold. Note that we apply up to 1 of each transformation, thus we also get the original string. Also, note that the two transformations cannot modify overlapping substrings of the input; for example, $T_{\text{swap}}$ did not swap the *ea* in *pleasant*.

### 4. Augmented Abstract Adversarial Training

In this section, we describe our abstract training technique, A3T, which combines augmentation and abstraction.

Recall the adversarial training objective function, Eq. (2). The difficulty in solving this objective is the inner maximization objective: $\max_{\mathbf{z} \in R(x)} \mathcal{L}(\mathbf{z}, y, \theta)$, where the perturbation space $R(x)$ can be intractably large to efficiently enumerate, and we therefore have to resort to approximation.

We begin by describing two approximation techniques and then discuss how our approach combines and extends them.

**Augmentation (search-based) techniques** We call the first class of techniques augmentation techniques, since they search for a worst-case sample in the perturbation space $R(x)$ with which to augment the dataset. The naïve way is to simply enumerate all points in $R(x)$—our specifications induce a finite perturbation space, by construction. Unfortunately, this can drastically slow down the training. For example, suppose $T$ defines a transformation that swaps two adjacent characters. On a string of length $N$, the specification $(T, 2)$ results in $O(N^2)$ transformations.

An efficient alternative, HotFlip, was proposed by Ebrahimi et al. (2018). HotFlip efficiently encodes a transformation $T$ as an operation over the embedding vector and approximates the worst-case loss using a single forward and backward pass through the network. To search through a set of transformations, HotFlip employs a beam search of some size $k$.
We now show how to define the abstraction of a perturbation space $S(x)$ defined by a specification $S = \{(T_1, \delta_1), \ldots, (T_n, \delta_n)\}$. Our approach generalizes that of Huang et al. (2019) to length-preserving transformations, i.e., ones that lengthen or shorten a string by up to $ \delta_i$ for all $i$.

**Algorithm 1: A3T**

**Input:** $S = \{(T_1, \delta_1), \ldots, (T_n, \delta_n)\}$ and point $(x, y)$

**Output:** worst-case loss

Split $S$ into $S_{\text{aug}}$ and $S_{\text{abs}}$ and return

$$\max_{\mathbf{z} \in \text{augment}_k(S_{\text{aug}}, x)} \mathcal{L}^{\text{perturb}}(\tilde{z}, y, \theta) \text{ s.t. } \tilde{z} = \text{abstract}(S_{\text{abs}}, \mathbf{z})$$

where the length of every string in $S(x)$ is the same as the length of the original string $x$. The approach of Huang et al. (2019) targeted the special case of single-character substitutions.

**Single transformation case** We first demonstrate the case of a single length-preserving transformation, $S = \{(T, \delta)\}$. Henceforth we assume that each element of a string $x$ is a real value, e.g., the embedding of a character or word. At a high level, our abstraction computes the convex hull that contains all the points in $T(x)$ (we use $T(x)$ as a short hand for the perturbation space obtained by applying $T$ to $x$ exactly once) and then scales this convex hull by $\delta$ to account for the cases in which $T$ is applied up to $\delta$ times. We begin by computing all points in $T(x)$. Let this set be $x_0, \ldots, x_m$. Next, for $i \in [1, m]$, we define the set of points

$$v_i = x + \delta \cdot (x_i - x).$$

We then construct the abstraction $\text{abstract}(S_{\text{abs}}, \mathbf{z})$ as the convex hull of the points $v_i$ and $x$. Observe that we only need to enumerate the space $T(x)$ obtained by applying $T$ once; multiplying by $\delta$ dilates the convex hull to include all strings that involve up to $\delta$ applications of $T$, i.e., $S(x)$. To propagate this convex hull through the network, we typically overapproximate it as a set of interval constraints, where each dimension of a string is represented by a lower and an upper bound. Interval constraints are easier to propagate through the network—requiring several forward passes linear in the length of the string—compared to arbitrary convex polyhedra, whose operations can be exponential in the number of dimensions (Cousot & Halbwachs, 1978).

**Example 5.** Consider the left side of Fig. 2. Say we have the string \texttt{ab} and the transformation \texttt{T} that can replace character \texttt{a} with \texttt{z}, or \texttt{b} with \texttt{g}—mimicking spelling mistakes, as these characters are adjacent on a QWERTY keyboard. The large shaded region is the result of dilating \texttt{T} with $\delta = 2$, i.e., contains all strings that can be produced by applying \texttt{T} twice to the string \texttt{ab}, namely, the string \texttt{gz}.

**General case** We generalize the above abstraction process to the perturbation space $S = \{(T_1, \delta_1), \ldots, (T_n, \delta_n)\}$. First, we enumerate all the strings in $T_1(x) \cup \cdots \cup T_n(x)$.
We prove Theorem 1 in the Appendix. As with the single-transformation case, we can now construct an abstraction of the convex hull induced by $v_i$ and $x$ as a set of intervals and propagate it through the network.

Example 6. We illustrate the process of abstracting an input string $cc$ on the right of Figure 2 for the specification $\{(T_{\text{prev}}, 1), (T_{\text{suc}}, 1)\}$, where $T_{\text{prev}}$ maps one character to its preceding character in the alphabet order, e.g., $c$ with $b$, and $T_{\text{suc}}$ maps one character to its succeeding character in the alphabet order, e.g., $c$ with $d$. We enumerate all the points in $T_{\text{prev}}(cc) \cup T_{\text{suc}}(cc)$ and compute their convex hull, shown as the inner shaded region. This region includes all strings resulting from exclusively one application of $T_{\text{prev}}$ or $T_{\text{suc}}$. Next, we dilate the convex hull by $2 = 1 + 1$ times to include all the points in the perturbation space of $\{(T_{\text{prev}}, 1), (T_{\text{suc}}, 1)\}$. This is shown as the larger shaded region. Notice how this region includes $bd$ and $db$, which result from an application of $T_{\text{suc}}$ to the first character and $T_{\text{prev}}$ to the second character of the original string $cc$.

The following theorem states that this process is sound: produces an overapproximation of a perturbation space $S(x)$.

Theorem 1. For every specification $S$ and input $x$, the abstracted perturbation space abstract$(S, x)$ is an overapproximation of $S(x)$—i.e., $S(x) \subseteq \text{abstract}(S, x)$

We prove Theorem 1 in the Appendix.

5. Experiments

In this section, we evaluate A3T by answering the following research questions:

- RQ1: Does A3T improve robustness in rich perturbation spaces for character-level and word-level models?
- RQ2: How does the complexity of the perturbation space affect the effectiveness of A3T?

5.1. Experimental Setup

5.1.1. DATASETS AND MODELS

We use two datasets:

- AG News (Zhang et al., 2015) dataset consists of a corpus of news articles collected by Gulli (2005) about the four largest news topics. We used the online-available dataset\(^2\), which contains 30,000 training and 1,900 testing examples for each class. We split the first 4,000 training examples for validation.

- SST2 (Socher et al., 2013) is the Stanford Sentiment Treebank dataset that consists of sentences from movie reviews and human annotations of their sentiment. The task is to predict the sentiment (positive/negative) of a given sentence. We used the dataset provided by TensorFlow\(^3\). The dataset contains 67,349 training, 872 validation, and 1,821 testing examples for each class.

For the AG dataset, we trained a smaller character-level model than the one used in Huang et al. (2019) but kept the number of layers and the data preprocessing the same. For the SST2 dataset, we trained a word-level model and a character-level model. We used the same models in Huang et al. (2019), also following their setup. We provide the details of the setups in the Appendix.

5.1.2. PERTURBATIONS

Our choice of models allows us to experiment with both character-level and word-level perturbations. We evaluated A3T on six perturbation spaces constructed using the seven individual string transformations in Table 1.

For the character-level model on dataset AG, we used the following specifications: $\{(T_{\text{SwapPair}}, 2), (T_{\text{SubAdj}}, 2)\}$, $\{(T_{\text{Del}}, 2), (T_{\text{SubAdj}}, 2)\}$, and $\{(T_{\text{InsAdj}}, 2), (T_{\text{SubAdj}}, 2)\}$. For example, the first specification mimics the combination of two spelling mistakes: swap two characters up to twice and/or substitute a character with an adjacent one on the keyboard up to twice.

For the word-level model on dataset SST2, we used the following specifications: $\{(T_{\text{DelStop}}, 2), (T_{\text{SubSyn}}, 2)\}$, $\{(T_{\text{DelStop}}, 2), (T_{\text{Dup}}, 2)\}$, and $\{(T_{\text{DelStop}}, 2), (T_{\text{Dup}}, 2), (T_{\text{SubSyn}}, 2)\}$. For example, the first specification removes stop words up to twice and substitutes up to twice words with synonyms.

For the character-level model on dataset SST2, we used the following specifications: $\{(T_{\text{SwapPair}}, 1), (T_{\text{SubAdj}}, 1)\}$, $\{(T_{\text{Del}}, 1), (T_{\text{SubAdj}}, 1)\}$, and $\{(T_{\text{InsAdj}}, 1), (T_{\text{SubAdj}}, 1)\}$. For example, the first specification mimics the combination of two spelling mistakes: swap two characters and/or substitute a character with an adjacent one on the keyboard.

For the character-level model on AG dataset, we considered the perturbations to be applied to a prefix of an input string, namely, a prefix length of 35 for $\{(T_{\text{SwapPair}}, 2), (T_{\text{SubAdj}}, 2)\}$, a prefix length of 30 for $\{(T_{\text{Del}}, 2), (T_{\text{SubAdj}}, 2)\}$, and $\{(T_{\text{InsAdj}}, 2), (T_{\text{SubAdj}}, 2)\}$. For the character-level model on SST2 dataset, we considered

\(^2\)This is the website describing the dataset: https://github.com/mhjabreel/CharCnn_Keras/tree/master/data/ag_news_csv.

\(^3\)This is the website describing the dataset: https://www.tensorflow.org/datasets/catalog/glue.
Robustness to String Transformations via Augmented Abstract Training

Table 1. String transformations to construct the perturbation spaces for evaluation.

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
<th>Training</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>${T}_{\text{SwapPair}}$</td>
<td>swap a pair of two adjacent characters</td>
</tr>
<tr>
<td></td>
<td>${T}_{\text{Del}}$</td>
<td>delete a character</td>
</tr>
<tr>
<td></td>
<td>${T}_{\text{InsAdj}}$</td>
<td>insert to the right of a character one of its adjacent characters on the keyboard</td>
</tr>
<tr>
<td></td>
<td>${T}_{\text{SubAdj}}$</td>
<td>substitute a character with an adjacent character on the keyboard</td>
</tr>
<tr>
<td>WORD</td>
<td>${T}_{\text{DelStop}}$</td>
<td>delete a stop word</td>
</tr>
<tr>
<td></td>
<td>${T}_{\text{Dup}}$</td>
<td>duplicate a word</td>
</tr>
<tr>
<td></td>
<td>${T}_{\text{SubSyn}}$</td>
<td>substitute a word with one of its synonyms</td>
</tr>
</tbody>
</table>

Table 2. Qualitative examples. The vanilla models correctly classify the original samples but fail to classify the perturbed samples.

<table>
<thead>
<tr>
<th>Prediction</th>
<th>A character-level sample and a perturbed sample in ${(T_{\text{SwapPair}}, 2), (T_{\text{SubAdj}}, 2)}$ of AG dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sci/Tech</td>
<td>ky. company wins grant to study peptides (ap) ap - a company founded by a chemistry researcher ...</td>
</tr>
<tr>
<td>World</td>
<td>yk. compnay wins granf to st8dy peptides (ap) ap - a company founded by a chemistry researcher ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prediction</th>
<th>A word-level sample and a perturbed sample in ${(T_{\text{DelStop}}, 2), (T_{\text{SubSyn}}, 2)}$ of SST2 dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive</td>
<td>a dream cast of solid female talent who build a seamless ensemble .</td>
</tr>
<tr>
<td>Negative</td>
<td>≠ dreaming casting of solid female talent who build ≠ seamless ensemble .</td>
</tr>
</tbody>
</table>

Perturbations with $\delta = 1$ but allow the perturbations to be applied to the whole input string. We made these restrictions because one cannot efficiently evaluate the exhaustive accuracy with larger $\delta$, due to the combinatorial explosion of the size of the perturbation space.

5.1.3. Training Methods

We implement and compare the following training methods.

**Normal training** is the vanilla training method (Eq. (1)) that minimizes the cross entropy between predictions and target labels. This method does not use the perturbation space and does not attempt to train a robust model.

**Random augmentation** performs adversarial training (Eq. (2)) using a weak adversary that simply picks a random perturbed sample from the perturbation space.

**HotFlip augmentation** performs adversarial training (Eq. (2)) using the HotFlip (Ebrahimi et al., 2018) attack to solve the inner maximization problem.

**A3T** is our technique that can be implemented in various ways. For our experiments, we made the following choices. First, we manually labeled which transformations in $S$ are explored using augmentation and which ones are explored using abstract interpretation (the third column in Table 1). Second, we implemented two different ways of performing data augmentation for the transformations in $S_{\text{aug}}$: (1) A3T(HotFlip) uses HotFlip to find the worst-case samples for augmentation, while (2) A3T(search) performs an explicit search through the perturbation space to find the worst-case samples for augmentation. Finally, we used DiffAI (Mirman et al., 2018) to perform abstract training for the transformations in $S_{\text{abs}}$, using the intervals abstraction.

In all augmentation training baselines, and A3T, we also adopt a curriculum-based training method (Huang et al., 2019; Gowal et al., 2019) which uses a hyperparameter $\lambda$ to weigh between normal loss and maximization objective in Eq. (2).

5.1.4. Evaluation Metrics

**Normal accuracy** is the vanilla accuracy of the model on the test set.

**HotFlip accuracy** is the adversarial accuracy of the model with respect to the HotFlip attack, i.e., for each point in the test set, we apply the HotFlip attack and test if the classification is still correct.

**Exhaustive accuracy** (Eq 3) is the worst-case accuracy of the model: a prediction on $(x, y)$ is considered correct if and only if all points $z \in S(x)$ lead to the correct prediction.

By definition, HotFlip accuracy is an upper bound on exhaustive accuracy.

---

4We consider this choice of split to be a hyperparameter.
Table 3. Experiment results for the three perturbations on the character-level model on AG dataset. We show the normal accuracy (Acc.), HotFlip accuracy (HF Acc.), and exhaustive accuracy (Exhaustive) of five different training methods.

<table>
<thead>
<tr>
<th>Training</th>
<th>({(T_{\text{SwapPair}}, 2), (T_{\text{SubAdj}}, 2)})</th>
<th>({(T_{\text{Del}}, 2), (T_{\text{SubAdj}}, 2)})</th>
<th>({(T_{\text{InsAdj}}, 2), (T_{\text{SubAdj}}, 2)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>87.5 71.5 60.1</td>
<td>87.5 79.0 62.5</td>
<td>87.5 79.1 59.0</td>
</tr>
<tr>
<td>Random Aug.</td>
<td>87.5 75.7 68.2 [+8.1]</td>
<td>87.4 81.3 69.4 [+6.9]</td>
<td>87.8 81.2 69.7 [+10.7]</td>
</tr>
<tr>
<td>HotFlip Aug.</td>
<td>86.6 85.7 84.9 [+24.8]</td>
<td>85.8 84.9 82.7 [+20.2]</td>
<td>86.8 85.9 82.6 [+23.6]</td>
</tr>
<tr>
<td>A3T(HotFlip)</td>
<td>86.4 86.4 86.4 [+26.3]</td>
<td>87.2 87.1 85.7 [+23.2]</td>
<td>87.4 87.4 85.5 [+26.5]</td>
</tr>
<tr>
<td>A3T(search)</td>
<td>86.9 86.8 86.8 [+26.7]</td>
<td>87.6 87.4 86.2 [+23.7]</td>
<td>87.9 87.8 86.5 [+27.5]</td>
</tr>
</tbody>
</table>

Table 4. Experiment results for the three perturbations on the word-level model on SST dataset.

<table>
<thead>
<tr>
<th>Training</th>
<th>({(T_{\text{DelStop}}, 2), (T_{\text{SubSyn}}, 2)})</th>
<th>({(T_{\text{DelStop}}, 2), (T_{\text{Dup}}, 2)})</th>
<th>({(T_{\text{DelStop}}, 2), (T_{\text{Dup}}, 2), (T_{\text{SubSyn}}, 2)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>82.4 68.9 64.4</td>
<td>82.4 55.8 47.9</td>
<td>82.4 54.8 42.4</td>
</tr>
<tr>
<td>Random Aug.</td>
<td>80.0 70.0 66.0 [+1.6]</td>
<td>81.5 54.2 49.7 [+1.8]</td>
<td>81.0 56.1 46.2 [+3.8]</td>
</tr>
<tr>
<td>HotFlip Aug.</td>
<td>80.8 74.4 68.3 [+3.9]</td>
<td>80.8 68.7 56.0 [+8.1]</td>
<td>81.2 69.0 51.0 [+8.6]</td>
</tr>
<tr>
<td>A3T(HotFlip)</td>
<td>80.2 73.5 70.2 [+5.8]</td>
<td>79.9 69.7 57.7 [+9.8]</td>
<td>78.8 68.1 55.1 [+12.7]</td>
</tr>
<tr>
<td>A3T(search)</td>
<td>79.9 74.4 71.2 [+6.8]</td>
<td>79.0 70.7 62.7 [+14.8]</td>
<td>77.7 69.8 59.8 [+17.4]</td>
</tr>
</tbody>
</table>

5.2. Evaluation Results

RQ1: Increase in robustness. We show the results for the selected perturbation spaces on character-level and word-level models in Tables 3, 4, and 5, respectively.

Compared to normal training, the results show that both A3T(HotFlip) and A3T(search) increase the exhaustive accuracy and can improve the robustness of the model. A3T(HotFlip) and A3T(search) also outperform random augmentation and HotFlip augmentation. In particular, A3T(search) has exhaustive accuracy that is on average 20.3 higher than normal training, 14.6 higher than random augmentation, and 6.7 higher than HotFlip augmentation.

We also compared A3T to training using only abstraction (i.e., all transformations in \(S_{\text{abs}}\) are also in \(S_{\text{abs}}\)) for the specification \(\{(T_{\text{SwapPair}}, 2), (T_{\text{SubAdj}}, 2)\}\) on AG dataset and \(\{(T_{\text{SwapPair}}, 1), (T_{\text{SubAdj}}, 1)\}\) on SST2 dataset (not shown in Tables 3, 4, and 5); this is the only specification that can be fully trained abstractly since it only uses length-preserving transformations. Training using only abstraction yields an exhaustive accuracy of 86.9 for \(\{(T_{\text{SwapPair}}, 2), (T_{\text{SubAdj}}, 2)\}\) on AG dataset, which is similar to the exhaustive accuracy of A3T(HotFlip) (86.4) and A3T(search) (86.8). However, training using only abstraction yields an exhaustive accuracy of 47.0 for \(\{(T_{\text{SwapPair}}, 1), (T_{\text{SubAdj}}, 1)\}\) on SST2 dataset, which is better than the one obtained using normal training, but much lower than the exhaustive accuracy of A3T(HotFlip) and A3T(search). Furthermore, the normal accuracy of the abstraction technique on SST2 dataset drops to 58.8 due to the over-approximation of the perturbation space while A3T(HotFlip) (73.6) and A3T(search) (70.2) retain high normal accuracy.

To answer RQ1, A3T yields models that are more robust to complex perturbation spaces than those produced by augmentation and abstraction techniques. This result holds for both character-level and word-level models.

RQ2: Effects of size of the perturbation space. In this section, we evaluate whether A3T can produce models that are robust to complex perturbation spaces. We fix the word-level model A3T (search) trained on \(\{(T_{\text{DelStop}}, 2), (T_{\text{SubSyn}}, 2)\}\). Then, we test this model’s exhaustive accuracy on \(\{(T_{\text{DelStop}}, \delta_1), (T_{\text{SubSyn}}, \delta_2)\}\) (Figure 3(a)) and \(\{(T_{\text{DelStop}}, 2), (T_{\text{SubSyn}}, \delta_2)\}\) (Figure 3(b)), where we vary the parameters \(\delta_1\) and \(\delta_2\) between 1 and 4, increasing the size of the perturbation space. (The Appendix contains a more detailed evaluation with different types of transformations.) We only consider word-level models because computing the exhaustive accuracy requires us to enumerate all the elements in the perturbation space. While enumeration is feasible for word-level transformations (e.g., the perturbation space of \(\{(T_{\text{DelStop}}, 4), (T_{\text{SubSyn}}, 2)\}\) for a string with 56 tokens contains at most 68,002 perturbed samples), enumeration is infeasible for character level transformations (e.g., the perturbation space of \(\{(T_{\text{Del}}, 4), (T_{\text{SubAdj}}, 2)\}\) on a prefix of length 30 of string with 300 characters contains 7,499,469 perturbed samples, and the perturbation space of \(\{(T_{\text{Del}}, 2), (T_{\text{SubAdj}}, 1)\}\) for a string with 300 characters...
Table 5. Experiment results for the three perturbations on the character-level model on SST2 dataset.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>77.0</td>
<td>36.5</td>
<td>Exhaustive</td>
<td>77.0</td>
<td>50.7</td>
<td>25.8</td>
<td>77.0</td>
<td>51.0</td>
<td>24.4</td>
</tr>
<tr>
<td>Random Aug.</td>
<td>75.6</td>
<td>47.1</td>
<td>28.2 [+5.2]</td>
<td>75.7</td>
<td>56.4</td>
<td>29.3 [+3.5]</td>
<td>74.5</td>
<td>57.0</td>
<td>33.8 [+9.4]</td>
</tr>
<tr>
<td>HotFlip Aug.</td>
<td>71.4</td>
<td>63.9</td>
<td>34.8 [+11.8]</td>
<td>76.6</td>
<td>67.1</td>
<td>38.0 [+12.2]</td>
<td>76.1</td>
<td>70.4</td>
<td>33.4 [+9.0]</td>
</tr>
<tr>
<td>A3T(HotFlip)</td>
<td>73.6</td>
<td>54.8</td>
<td>35.2 [+12.2]</td>
<td>75.3</td>
<td>58.2</td>
<td>32.9 [+7.1]</td>
<td>72.4</td>
<td>66.3</td>
<td>44.7 [+20.3]</td>
</tr>
<tr>
<td>A3T(search)</td>
<td>70.2</td>
<td>57.1</td>
<td>48.7 [+15.7]</td>
<td>72.5</td>
<td>62.5</td>
<td>44.8 [+19.0]</td>
<td>71.6</td>
<td>65.0</td>
<td>55.2 [+30.8]</td>
</tr>
</tbody>
</table>

Figure 3. The exhaustive accuracy of \(\{T_{\text{DelStop}}, \delta_1\}, \{T_{\text{SubSyn}}, \delta_2\}\) varying the parameters \(\delta_1\) (left) and \(\delta_2\) (right) between 1 and 4.

6. Conclusion, Limitations, and Future Work

We presented an adversarial training technique, A3T, combining augmentation and abstraction techniques to achieve robustness against programmable string transformations in neural networks for NLP tasks. In the experiments, we showed that A3T yields more robust models than augmentation and abstraction techniques.

We foresee many future improvements to A3T. First, A3T cannot currently generalize to RNNs because its abstraction technique can only be applied to models where the first layer is an affine transformation (e.g., linear or convolutional layer). Applying A3T to RNNs will require designing new abstraction techniques for RNNs. Second, we manually split \(S\) into \(S_{\text{aug}}\) and \(S_{\text{abs}}\). Performing the split automatically is left as future work. Third, A3T(search) achieves the best performance by looking for the worst-case perturbed sample in the perturbation space of \(S_{\text{aug}}\) via enumeration. In some practical settings, \(S_{\text{aug}}\) might induce a large perturbation space, and it might be best to use A3T(HotFlip) instead. Fourth, we choose HotFlip and interval abstraction to approximate the worst-case loss in our experiments, but our approach is general and can benefit from new augmentation and abstraction techniques.

contains 20,252,321,116 perturbed samples!).

The exhaustive accuracy of A3T(HotFlip) and A3T(search) decreases by 1.6% and 1.1%, respectively, when increasing \(\delta_1\) from 1 to 4, and decreases by 2.1% and 1.4%, respectively, when increasing \(\delta_2\) from 1 to 4. All other techniques result in larger decreases in exhaustive accuracy (\(\geq 1.7\%\) in \(\{T_{\text{DelStop}}, \delta_1\}, \{T_{\text{SubSyn}}, \delta_2\}\) and \(\geq 4.2\%\) in \(\{T_{\text{DelStop}}, 2\}, \{T_{\text{SubSyn}}, \delta_2\}\)).

To answer RQ2, even in the presence of large perturbation spaces A3T yields models that are more robust than those produced by augmentation techniques.
Acknowledgements

We would like to thank Samuel Drews for his feedback, as well as the reviewers. This work is supported by the National Science Foundation grants CCF-1704117, CCF-1918211, and Facebook research award Probability and Programming.

References


Robustness to String Transformations via Augmented Abstract Training


