2019

Improved neural machine translation systems for low resource correction tasks

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

IMPROVED NEURAL MACHINE TRANSLATION SYSTEMS FOR LOW RESOURCE CORRECTION TASKS

by

JACOB HARER

B.S., Duke University, 2011
M.S., North Carolina State University, 2012

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2019

Approved by

First Reader

Sang Chin, PhD
Research Professor of Computer Science

Second Reader

Chris Reale, PhD
Member of the Technical Staff, Charles Stark Draper Laboratory

Third Reader

Margrit Betke, PhD
Professor of Computer Science

Fourth Reader

Derry Wijaya, PhD
Assistant Professor of Computer Science

IMPROVED NEURAL MACHINE TRANSLATION SYSTEMS FOR LOW RESOURCE CORRECTION TASKS

JACOB HARER

Boston University, Graduate School Of Args and Sciences, 2019

Major Professor: Sang Chin, PhD
Research Professor of Computer Science

ABSTRACT

Recent advances in Neural Machine Translation (NMT) systems have achieved impressive results on language translation tasks. However, the success of these systems has been limited when applied to similar low-resource tasks, such as language correction. In these cases, datasets are often small whilst still containing long sequences, leading to significant overfitting and poor generalization. In this thesis we study issues preventing widespread adoption of NMT systems into low resource tasks, with a special focus on sequence correction for both code and language. We propose two novel techniques for handling these low-resource tasks. The first uses Generative Adversarial Networks to handle datasets without paired data. This technique allows the use of available unpaired datasets which are typically much larger than paired datasets since they do not require manual annotation. We first develop a methodology for generation of discrete sequences using a Wasserstein Generative Adversarial Network, and then use this methodology to train a NMT system on unpaired data. Our second technique converts sequences into a tree-structured representation, and performs translation from tree-to-tree. This improves the handling of very long sequences since it reduces the distance between nodes in the network, and allows the network to take advantage of information contained in the tree structure to reduce overfitting.

Contents

List of Tables

List of Figures

List of Abbreviations

AST	.....	Abstract Syntax Tree
CFG	.....	Context Free Grammar
CNN	.....	Convolutional Neural Network
GAN	.....	Generative Adversarial Network
LSTM	.....	Long Short-Term Memory
NLM	.....	Neural Language Modeling
NLP	.....	Natural Language Processing
NMT	.....	Neural Machine Translation
PCFG	.....	Probabilistic Context Free Grammar
PMF	.....	Probability Mass Function
RNN	.....	Recurrent Neural Network
Seq2Seq	.....	Sequence to Sequence
Tree2Tree	.....	Tree to Tree
WGAN	.....	Wasserstein Generative Adversarial Network

Chapter 1

Introduction

Automated translation has become an increasingly studied domain among machine learning researchers. Neural network based translation systems, or neural machine translation (NMT) systems, cover a growing set of machine learning methods which translate data from one domain to another. These methods include prominent, state of the art, results for language translation [1, 2, 3, 4, 5, 6, 7]. These successes have prompted researches to apply NMT systems to other similar tasks, such as correction. However, adoption of NMT into real world applications beyond language translation has been slow, struggling to compete with more traditional, non-neural, approaches, and convolutional neural network based methods. In this thesis, we study the application of NMT to correction tasks, focusing on methods to improve the performance of these systems.

The ability to communicate across language through translation has been of interest to scholars since antiquity, with the first major translation of the western world widely regarded to be the translation of the Hebrew bible into Greek in the 3rd century [8]. The need for translations has only grown throughout the ages. In todays multicultural, and by extension multi-lingual, world, the need for accurate translation cannot be understated. This same need also extends to language correction, which helps to produce grammatically correct and understandable text. Language correction is of particular interest in scientific domains, where the desire to have literature published in a common language has led to a significant rise in publications by non-native speakers.

The study of Natural Language Processing, and as a sub-field the automation of translation and correction systems, has been highly studied by computer science faculty for some time. Interest in translation systems dates back to the 1950's, where growing work in mainstream linguistics lead researchers to produce the first successful systems [9]. Since then, improvements in processing of grammars and semantics has lead to many very impressive systems including database systems such as LIFER/LADDER [10], semantic based systems such as KL-ONE [11], and probability based systems such as Hidden Markov Models [12].

The resurgence of neural networks in machine learning research has also lead to an increased

interest in NMT systems. These systems use a neural network to model the probability of generating an output sequence given an input sequence. Recent NMT systems have outpaced prior methods for language translation [5, 6, 7], achieving significant improvements on many benchmark machine translation tasks. However, the success of these systems often relies upon very large datasets, which makes them difficult to use for less common, and lower resource tasks.

This thesis focuses on the challenge of applying recent NMT systems to correction tasks, which are often significantly lower resource than their translation counterparts. It attempts to improve performance on these systems in two ways: a) training the system on much larger unpaired datasets using a generative adversarial network, b) using tree representations of the data to improve the networks understanding of the underlying grammar.

1.1 Machine Translation on Low Resource Data

NMT systems model the probability of outputs based on an input. Real data, such as language, is incredibly complex, which makes accurately modeling probabilities a difficult task. This complexity can be a major issue when dealing with low resource datasets, which often are not large enough to be representative of the distribution from which they were drawn. This means that a model trained from scratch on a low resource dataset will not be able to accurately model the distribution of real data, and therefore has limited performance regardless of the model chosen.

Most current techniques for handling low resource datasets involve use of additional data. Additional data is most commonly used for pre-training, or transfer learning, where a network is first trained on a larger similar dataset and then used as the initial state for a network trained on the low resource dataset. A popular example of this for NMT systems is monolingual pre-training, where the network is first trained as a language model on a large monolingual dataset [13].

Work in this thesis attempts to give our NMT systems additional information by supplying them with a more informative representation of the data. The Tree Transformer method, explained in chapter 5 uses prior work on sequence to tree parsers to produce tree representations of the data. These tree representations contain information about the grammatical structure of the data, giving the network the ability to understand the grammar and thus improving performance on low-resource tasks.

Table 1.1: Approximate dataset sizes for translation and correction datasets

Dataset	Description	Size
WMT 2014	English to German	4M
WMT 2018	English to German	41M
CONLL 2014	English Language Correction	50K
LANG 8	English Language Correction	1M (300K Corrections)
SATE IV	C/C++ Code Correction	50K

1.2 Correction

In addition to the myriad of new translation systems, correction based systems have also become more prevalent. Neural network based correction systems follow the same design as translations systems by translating from incorrect sequences to correct ones.

One of the most common usages for correction systems is the task of grammar error correction (GEC), which seeks to automatically correct grammatically incorrect sentences into correct ones. Successful approaches to GEC could be very useful to non-native speakers writing in a foreign language. Additional applications of these systems includes replacement of current spelling and grammar checkers, and in interactive teaching of foreign languages.

Automatic correction of code is also an increasingly popular domain, including both the repair of bugs and security vulnerabilities. Current methods for detection and repair of vulnerabilities in code are time consuming and expensive, requiring companies to hire large numbers of verification engineers. Automatic code correction could assist, or replace, these engineers by identifying and repairing security vulnerabilities in code prior to release, potentially avoiding millions of dollars of reparation costs.

The datasets currently available for language and code correction systems are small, and therefore low-resource. Table 1.1 shows the relative sizes of the WMT language translation dataset to correction datasets used in this thesis. The most recent WMT dataset for English to Germain translation contains 41 Million sentence pairs. In contrast, CONLL 2014, which is still the most commonly used English correction dataset, contains around 50K pairs. For correction datasets to be comparable in size to their translation counterparts they would need to be increased in size by several orders of magnitude.

1.3 Paired vs Unpaired data

Typical training data for NMT systems is provided in pairs. Each pair provides an input sequence and the desired output sequence it should be translated into. This allows training by providing the

system with the input sequence and then computing the error between the output of the system and the desired output. Conversely to paired data, unpaired data does not provide links between input and output data-points. Instead, unpaired data consists simply of sets of input and output data. This makes the problem far more difficult and requires an alternative approach.

Interest in systems which utilize unpaired data is motived by the fact that unpaired data is generally much easier to obtain than paired data. Paired data typically requires some form of hand annotation. For example, the language correction dataset CONLL 2014 was created by having three reviewers review a large quantity of essays and then pooling their corrections. This need for hand annotation makes the generation of large datasets challenging, leading to much smaller datasets for less common tasks, such as correction. However, generating unpaired data requires only that examples are labeled as either input or output data, i.e. either incorrect or correct in the case of correction. This can often be achieved using readily available automated methods, allowing the creation of very large datasets with relative ease.

1.4 Contributions

This thesis makes the following contributions

- We design a system which uses a Wasserstein Generative Adversarial Network (WGAN) with an added conditional loss functions to train a NMT system. With this system we train a network to successfully correct software vulnerabilities.

- We develop the Tree-Transformer, a transformer based architecture which operates on tree data, translating from Tree-to-Tree. We show improved performance using this system on both code correction and language correction tasks.

- We compare and contrast methods for the use of a WGAN (WGAN) in generation of discrete sequences, including novel architectures, regularization techniques, and handling of embeddings. Methods considered are compared using a dataset created from a Probabilist Context Free Grammar, which allows us to compute more robust metrics than are typical of GAN techniques.

- We rigorously prove that the Wasserstein-1 distance cannot be used to directly train a language model which outputs probabilities.

- We are the first to generate realistic text using a GAN. We train a system on the One Billion Word dataset with a novel use of pre-trained embeddings between generator and discriminator.

1.5 Thesis Outline

The rest of the thesis is organized as follows.

Chapter 2: Background. This chapter outlines background for the techniques used in the rest of the thesis. It contains details on notations used, neural language modeling and NMT systems, generative adversarial networks including Wasserstein generative adversarial networks, and transformer networks.

Chapter 3: Improved Generation of Discrete Sequences using Wasserstein Generative Adversarial Networks. This chapter discusses the issues with, and provides a methodology for, using a GAN to generate sequences of discrete data. It starts by discussing difficulties with using a WGAN based system to train a language model and proves that minimizing the wasserstein-1 distance to train a NMT system does not produce correct probabilities. It then provides alternative approaches to this system. Using a generated dataset, it compares these alternatives, different generator and discriminator architectures, and several novel regularization techniques. Finally, using the best architecture from these comparisons we train a network to output realistic text on the one billion word dataset.

Chapter 4: Learning to Repair Software Vulnerabilities with Generative Adversarial Networks. This chapter builds on chapter 3 to train a NMT system using a WGAN. The chapter first outlines the GAN based system used, which includes a NMT system as the generator. Next, it adds novel loss functions which serves to condition the output of the NMT system on its input. The chapter then outlines the full system, including autoencoder pre-training and curriculum learning procedures. This system is then used to train a network which can correct security vulnerabilities in code.

Chapter 5: Tree-Transformer: A Transformer-Based Method for Correction of Tree-Structured Data. This chapter develops the tree-transformer system, which uses tree structured data to improve performance of NMT systems. The chapter provides the necessary modification to the popular transformer architecture for use on tree-structured data, including the design of a tree-convolution block which gives nodes access to the tree structure. The tree-transformer is then used on both a code correction dataset and two benchmark grammar correction datasets. For all three datasets we show improved performance over past NMT systems.

Chapter 6: Conclusion. This chapter summarizes the contributions, strengths, and weaknesses of the proposed methods from the previous chapters.

Chapter 2

Background

2.1 Variable Notation

This thesis primarily deals with sequential data. The total length of a sequence is represented with an uppercase T, while individual time-steps are represented using the lowercase t. Although sequences do not have to be over time, for example text is over a series of words, the convention is to refer to steps along the sequence dimension as time-steps, and represent them using t. To avoid confusion with probability notation both vectors and sequences of vectors are represented using lower case. The notation used is as follows:

- T - total number of time-steps for a sequence
- t - a particular time-step for a sequence, for example $\sum_{t=0}^{T-1} x_t$
- x_t - the tth vector in the sequence of vectors x.
- x - a sequence of vectors x_t, i.e. $[x_0, x_1, ...x_{T-1}]$

Many previous works use x as an input vector and y as the desired output vector. Unfortunately the names adopted for NMT are not compatible those for GANs. As such, we adopt the following naming convention:

- x - network input
- $\mathbb{P}(X)$ - distribution of network input
- r - desired output
- $\mathbb{P}(R)$ - distribution of desired output
- y - network output, sequence of probability vectors
- $\bar{r}$ - candidate vector for r, generally created by sampling each y_t
- z - sequence of noise vectors

2.2 Recurrent Neural Networks

RNN's handle sequences of arbitrary length by utilizing a recursive process. At each step t, they maintain a state vector, s_t, based on the input for time step t, x_t, and the previous state vector, s_{t-1}:

$$s_t = \sigma(x_t U + s_{t-1} W), \tag{2.1}$$

where σ is a non-linearity, commonly a sigmoid or hyperbolic tangent. Unfortunately, these non-linearities are a primary cause of the gradient vanishing problem, since their derivative with respect to their inputs is maximally 0.25 for sigmoid, and 1.0 for hyperbolic tangent. Thus gradients propagated through these non-linearities will generally shrink [14]. This finding prompted the creation of the Long Short-Term Memory unit (LSTM) [15], which has become the defacto standard for RNN's, written as:

$$\begin{aligned}
f_t &= \sigma(x_t U^f + s_{t-1} W^f), \\
i_t &= \sigma(x_t U^f + s_{t-1} W^f), \\
o_t &= \sigma(x_t U^f + s_{t-1} W^f), \\
u_t &= tanh(x_t U^c + s_{t-1} W^c), \\
c_t &= f_t \odot c_{t-1} + i_t \odot u_t, \\
s_t &= o_t \odot tanh(c_t).
\end{aligned}$$

LSTM modifies equation 2.1 with the additional hidden state c_t. Key to combating the gradient vanishing problem, the transition from c_{t-1} to c_t does not involve a non-linearity, and instead is only modified through element wise multiplication. Access to c_t is controlled by two gates f_t and i_t. f_t (the forget gate) controls to what extent the previous state c_{t-1} is retained, while i_t (the input gate) updates c_t based on the computations in u_t. Information is extracted from c_t using the output gate, o_t. Thanks to this setup, information can be written into the c state and extracted many time steps later, only passing through two non-linearities. In theory, this lack on non-linearities in the path of c_t gives LSTM's the ability to handle temporal dependencies of an arbitrary length. However, in practice computational difficulties often involve frequent modifications to c_t, with inputs and outputs both passing through a non-linearity. LSTMs allow handling of sequences which are far longer than default RNNs, but the gradient vanishing problem is still an issue for LSTMs when sequence lengths are in the tens to hundreds.

For simplicity, RNNs (including LSTMs) are often represented as a function of their input and

their past hidden state, or $s_t = f(x_t, s_{t-1})$. We will use this notation going forward.

2.3 Neural Language Modeling

Neural language models (NLMs) use a neural network to model language as a conditional distribution over a fixed vocabulary. NLMs approximate a real probability distribution $\mathbb{P}(R)$ given a set of samples r, which are assumed to be drawn independently from $\mathbb{P}(R)$. $\mathbb{P}(R)$ can be factored as

$$\mathbb{P}(R) = \prod_{t=0}^{T} \mathbb{P}(R_t = r_t | r_{<t}). \tag{2.2}$$

The conditional distribution $\mathbb{P}(R)$ is typically modeled using a RNN. The RNN produces a sequence of probability vectors, y_t, which each approximate $\mathbb{P}(R_t)$. Put more concretely, we desire $y_{t_i} = \mathbb{P}(R_t = r_{ti} | r_{<t}) \quad \forall t, i$, where r_{ti} is the ith possible output for the tth timestep. The use of an RNN allows conditioning each output, y_t, on previous inputs, $r_{<t}$, through the state vector s_t. This can be written as:

$$\begin{aligned} s_t &= f(r_{t-1}, s_{t-1}), \\ y_t &= \text{Softmax}(W s_t). \end{aligned} \tag{2.3}$$

In the above equation, s_t and y_t are conditioned on the previous time-step through the input r_{t-1}, and on all prior time-steps through the state s_{t-1}.

Approximating $\mathbb{P}(R_t)$ using y_t is usually achieved by minimizing the negative log likelihood (NLL) between the outputs y_t and the samples r_t, given as:

$$L(r_t, y_t) = -\sum_{i=1}^{N} \mathbb{P}(R_t = r_{ti}) \log(y_{ti}),$$

where the sum is over all of the N possible discrete outputs. $\mathbb{P}(R_t = r_{ti})$ is the probability of the ith possible output at time t and y_{ti} is the ith output of the probability vector y_t.

In equation 2.3, the input r_{t-1} can come from one of two places. If the desired r is known, for example during training, than r_{t-1} can be set to its given value. This technique is called teacher forcing [16], since r_{t-1} remains unchanged regardless of the value of y_{t-1}. In cases where r is unknown, such as testing, we can sample the previous outputs y_{t-1}, producing a candidate vector $\bar{r}_{t-1}$ and use this as an approximate value for r_{t-1}.

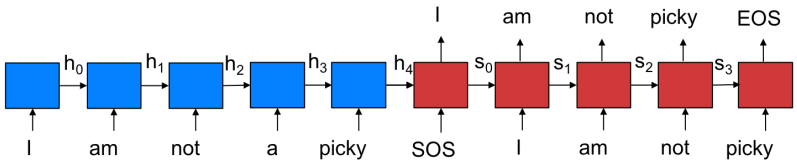


Figure 2·1: Sequence to Sequence network with encoder (blue) and decoder (red)

2.4 Neural Machine Translation

Neural Machine Translation (NMT) systems extend NLMs by also conditioning the produced output on an input sequence x. They model the conditional probability $\mathbb{P}(R = r|x)$ of transforming input sequence x into output sequence r. r can be factored it into a conditional distribution following equation 2.2 as:

$$\mathbb{P}(R = r|x) = \prod_{t=0}^{T} \mathbb{P}(R_t = r_t|r_{<t}, x).$$

NMT systems model this conditional distribution using an encoder and decoder approach, with an RNN for each part. The encoder reads the input sequence in order, producing each output as:

$$h_t = f_e(x_t, h_{t-1}).$$

The decoder follows the same procedure as NLMs by predicting the outputs $\mathbb{P}(R_t = r_t|r_{<t}, x)$ one at a time, following equation 2.3. Conditioning the decoders state, s_t, on the input, x, is achieved by setting the initial state, s_{-1}, to be the final encoder state h_T. This encoder and decoder setup are shown in figure 2·1

2.5 Attention

The vanilla NMT network, as described previously in 2.4, has significant issues with gradient vanishing. At worst case information must propagate from the first node in the encoder, x_0, to the last node in the decoder, y_T. Additionally, any information passed from encoder to decoder must be through the final encoder state h_T, which limits information transfer based on the state size. These issues are improved through the use of an attention mechanism [6, 7], shown in figure 2·2. Attention mechanisms provide each decoder state direct access to selected encoder states. Attention is computed as a weighted sum of values, v, with weights determined using a score function between queries, q, and keys, k. Typically in NMT, $k = v = h$ and $q = s$. Attention can be written as:

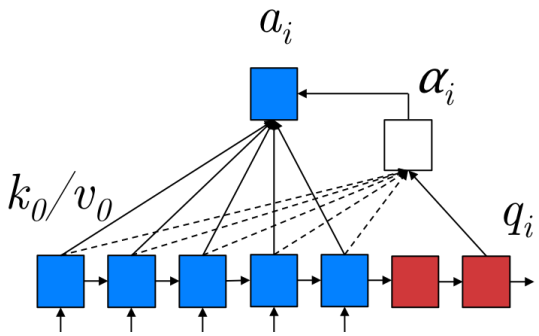


Figure 2·2: Attention mechanism attached to sequence to sequence network

$$\begin{aligned} \alpha_{ij} &= \frac{exp(score(q_i, k_j))}{\sum_{t=1}^{T} exp(score(q_i, k_t))}, \\ a_i &= \sum_{j=1}^{T} \alpha_{ij} v_j. \end{aligned} \tag{2.4}$$

In the case where every a_i can be computed in parallel this can be simplified using matrix notation as:

$$a = \text{Softmax}(score(Q, K))V.$$

Common choices for score functions are:

- General - $c_{it} = W_o \sigma(W_q q_i + W_k k_t)$
- Dot - $c_{it} = q_i^T k_t$

With attention the network output, y_t, is computed by combining the attention output with the current decoder state:

$$y_t = \text{Softmax}(W_2 \sigma(W_a a_i + W_s s_i)).$$

2.6 Transformer

The Transformer model of Vaswani et. al. [5] is an attention only system which forgoes the inter-layer connections of RNN networks in favor of attention. The transformer model has become increasingly popular because it trains very quickly. The training speedup of the Transformer over RNNs is largely due to the removal of the inter-layer RNN connections which allow all outputs of each decoder layers to be computed in parallel during training.

The transformer consists of three types of sub-layers, attention, self-attention, and feed-forward. Attention layers work similarly to those described in section 2.5, with queries from the decoder states and keys and values from the encoder states. The Transformer uses a scaled dot-product attention mechanism:

$$a = \text{softmax}(\frac{QK^T}{\sqrt{d^k}})V,$$

where d^k is the dimension of k. Self attention sub-layers are identical to attention layers except querys, keys, and values all come from the same location, either the encoder or decoder state. Finally, feed-forward layers combine two fully-connected layers with a non-linearity in between:

$$FFN(x) = W_2\text{ReLU}(W_1 x).$$

All sub-layers in the Transformer are wrapped with a layer-norm and residual connection:

$$\text{Layer-Norm}(x + \text{Sub-Layer}(x)).$$

Encoder layers are made up of a self-attention sub-layer followed by a fully-connected sub-layer. Decoder layers contain a self-attention, attention and then feed-forward sub-layers. The default Transformer model contains 6 encoder/decoder layers.

2.7 Generative Adversarial Networks

Generative Adversarial Networks (GANs) were originally proposed by Goodfellow et al [17] as a way to generate realistic images from random noise. This is achieved by framing learning as a 2-person minimax game between a generator $G(\cdot)$ and a discriminator $D(\cdot)$. The generator learns to generate realistic looking samples by minimizing the performance of the discriminator, whose goal in turn is to maximize its own performance on discriminating between generated samples $G(z)$ and real samples r.

The original GAN loss of Goodfellow et al. [17] is written as:

$$\begin{aligned}\mathcal{L}_{GAN}(D,G) =& \mathbb{E}_{r\sim\mathbb{P}(R=r)}[\log\ D(r)] \\ &+\mathbb{E}_{z\sim\mathbb{P}(Z=z)}[\log(1-D(G(z))].\end{aligned}$$

where the optimal generator is:

$$G^* = \arg\min_G \max_D \mathcal{L}_{GAN}(D,G).$$

This formulation is known to approximate the Jensen-Shannon (JS) divergence between the real distribution $\mathbb{P}(R)$ and the generated one $\mathbb{P}(G)$. However, when the support of the distributions of generated and real data do not overlap, the JS divergence is constant, and has a zero gradient, making the minimization of equation 2.7 by gradient decent impractical [18]. This is a prohibitive problem when dealing with discrete data because the distributions will never overlap since the support of the real samples r is a union of zero dimensional manifolds.

An alternative method was proposed by Arjovsky et al. [19]. Instead of JS, they approximate the Wasserstein-1 metric, or Earth-Movers (EM) distance. The Kantorovich Dual form for the Wasserstein-1 metric can be written as:

$$\mathcal{W}_1(\mathbb{P}(R), \mathbb{P}(Y)) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{r\sim\mathbb{P}(R)}[f(r)] - \mathbb{E}_{y\sim\mathbb{P}(Y)}[f(y)], \tag{2.5}$$

where the supremum is over all 1-Lipschitz functions.

Using a discriminator neural network, D, for f, and a generator network, G, to map from an input distribution Z to the generated distribution Y, gives us the easily commutable Wasserstein GAN (WGAN) loss function:

$$\mathcal{L}_{WGAN}(D, G) = \mathbb{E}_{r\sim\mathbb{P}(R)}[D(r)] - \mathbb{E}_{z\sim\mathbb{P}(Z)}[D(G(z))], \tag{2.6}$$

where the discriminator function D must be constrained to be 1-Lipschitz. Unlike JS, EM distance linearly decreases as $\mathbb{P}(G(z))$ approaches $\mathbb{P}(R)$, even for cases where their supports are disjoint.

Most recent work using WGANs have enforced the Lipschitz constraint on the discriminator through a regularization term with a gradient penalty (WGAN-GP), first introduced by Gulrajani et al. [20]. This approach penalizes the l2 norm of the gradient at the inputs to the discriminator, modifying the WGAN formulation of equation 2.6 as follows:

$$\begin{aligned}\mathcal{L}_{WGAN}(D, G) =& \mathbb{E}_{r\sim\mathbb{P}(R)}[D(r)] - \mathbb{E}_{z\sim\mathbb{P}(Z)}[D(G(z))] \\ &+\lambda\mathbb{E}_{\hat{y}\sim\mathbb{P}(\hat{Y})}[(\|\nabla_{\hat{y}} D(\hat{y})\|_2 - 1)^2].\end{aligned} \tag{2.7}$$

Ideally we would like to enforce this gradient penalty everywhere. However this is impractical. Typically, $\hat{y}$ is chosen linearly between pairs of points $G(z)$ and r.

Chapter 3

Improved Generation of Discrete Sequences using Wasserstein Generative Adversarial Networks

3.1 Introduction

Generative Adversarial Networks (GANs) [17] have proven to be a valuable tool for generating data using neural networks. They have enabled generation of very realistic and diverse samples of high-dimensional continuous data [21]. However, application of GAN methods to domains with discrete data, such as natural language, has proven more difficult. So far, GAN based language models have struggled to produce realistic examples of natural language, especially in comparison to other language modeling techniques [22, 23, 24, 25, 26].

GAN based methods for language generation are appealing because of the diversity of samples they produce. This diversity can be largely attributed to the fact that GANs match distributions rather than focusing on the reproduction of individual samples. In the context of language modeling, this means GAN methods should be far more robust to exposure bias than other neural network methods, making GANs appealing in cases where large training corpora are not available.

In addition to the advantages of GAN based approaches, other approaches are simply unusable for many applications. For example, several recent works have applied GANs in order to train Neuro Machine Translation models without paired data [27, 28, 29]. The absence of paired data means the desired translation for a given input is unknown, which precludes the use of traditional neural language modeling techniques. However, GAN approaches can overcome this limitation by using the adversarial objective of the GAN to match the distribution of translated outputs to the distribution of real sequencess in the output language. This approach requires an adversarial objective capable of handling discrete data, and would benefit heavily from improved GAN based training of language models. We will discuss this approach in detail in chapter 4.

This chapter addresses the problem of generating sequences using GANs, first by providing some theoretical background on the problem and then through a series of experimental results. The

chapter is broken up as follows. Section 3.2 studies the reasons why a GAN approach cannot be applied directly to a probabilistic neural language model, and rigorously proves this issue as it relates to the minimization of the Wasserstein-1 distance when using probabilistic outputs. It then provides some alternative approaches, their merits and their difficulties. Section 3.3 describes the architectures we used for our experiments, taking insight from section 3.2 and including several novel architectures, improved regularization, and handling of embeddings. Finally, section 3.4 contains experiment descriptions and results. We test our networks on two datasets. First on a dataset generated from a probabilistic context free grammar, which allows us to provide far more rigorous metrics than are typical of GAN experiments. Second, we test our most promising networks on real data taken from the Google One Billion Word dataset [26], and show that we are able to generate realistic looking English sentences of modest length. To the best of our knowledge we are the first to generate realistic looking text using a GAN without network pretraining.

3.2 Training a Language Model with WGAN

Recall from section 2.3 that neural language models typically output a probability vector at each timestep. This is useful in order to compute the probability of each output and for certain metrics like perplexity. We would like to train such a language model using a Wasserstein GAN, with the language model taking the place of the generator in equation 2.6. Such a model is shown in figure 3·1. Unfortunately, training a language model with a WGAN in this way does not work as desired, causing the outputs of the language model to not accurately represent the distribution of real data. This if formalized in the following lemma, with the proof included in appendix 3.A.

Lemma 3.2.1. *If r are samples from a discrete non-conditional distribution, $\mathbb{P}(R)$, over N finite values, and if y is a probability vector of length N, then minimizing $EM(r, y)$ is equivalent to solving for y such that $y = P(R = r)$ iff $P(R = r_k) = 1.0$ for some k or $P(R = r_i) = P(R = r_j)\ \forall\{i, j\}$.*

Lemma 3.2.1 proves that EM cannot be use to directly generate probability vectors from samples. Intuitively, the lemma holds because EM uses the $\ell 2$ distance metric between pairs of points. Since the gradient of $\ell 2$ distance is constant, the probability vector y is pulled more forcefully towards those real points, r, which occur with higher probability. A graphical view of this for a probability vector of length 2 is shown in figure 3·2

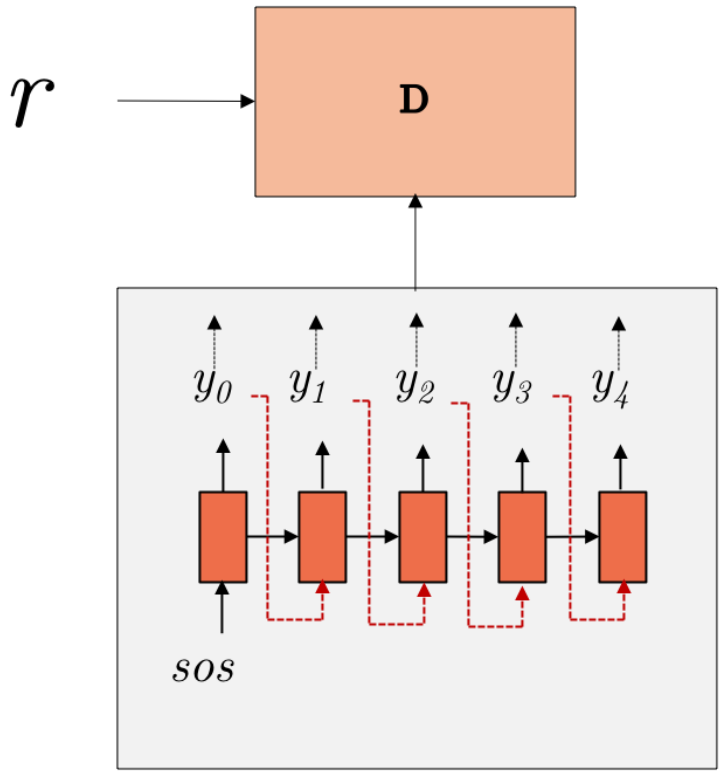


Figure 3·1: GAN system for training a language model. The language model is used as the generator to produce G(Z).

3.2.1 Alternative Approaches

Lemma 3.2.1 makes the naive approach of attaching the WGAN discriminator onto a standard neural language model not viable. However, several alternative approaches exist.

Policy Gradient

One approach is to sample from y at each time-step, yielding $\bar{r}_t$, and use this as input to the discriminator. This way the discriminator compares samples r to samples $\bar{r}$. Difficulty arises because the sampling operation is non differentiable, and therefore prevents updating the generator by using the gradient of equation 2.7. One way to address this is to use a policy gradient approach, which treats the output of the discriminator as a reward in a reinforcement learning setting [23, 30]. A major downside to this approach is that rewards are provided for entire sequences, meaning updates to the generator do not provide information on which parts of the output sequence the discriminator thinks

are incorrect. This results in long convergence times and low accuracies. Some success has been had with this approach, particularly when it comes to generation of discrete images [31]. However, generation of sequences seems to be a much harder problem, and to the best of our knowledge no one has been able to generate realistic looking text using policy gradient based methods without pretraining.

Added Noise

Another approach to address the problem in lemma 3.2.1 is to modify the language model to include an input noise vector, z_t, at each time-step, sampled uniformly from a Gaussian distribution. Including input noise means generator outputs, y_t, are determined both by the previous outputs and by the noise:

$$\begin{aligned} s_t &= \text{RNN}(\bar{r}_{t-1}, z_t, s_{t-1}), \\ y_t &= \text{Softmax}(W s_t). \end{aligned} \tag{3.1}$$

In theory, the inclusion of z_t allows the network to achieve an EM distance of 0 when y_t directly approximates r_t, i.e. y_t approximates sampling from $\mathbb{P}(R)$. The addition of the noise vector z_t can be thought of as moving the stochastic part of the network from the sampling operation to the noise vector z.

The approach of adding z_t is similar in nature to the GAN approach used to generate images, which utilizes a single noise vector z as input. However, one caveat with adding the noise vector z_t to a language model is that it no longer produces a probability vector as output, instead it directly approximates samples. I.e., we expect y_t to be close to a one-hot vector, and vary based on z_t. This makes it is impossible to accomplish some computations typical of language models. For example, we can no longer compute the probability, or perplexity, of a particular output. In many applications where directly training using NLL is not available this trade off may be acceptable.

Gumbel-Softmax

We would be remiss not to mention the Gumbel-Softmax approach of Jang et al. [32] and Maddison et al. [33]. This approach uses Gumbel noise to approximate sampling from an output probability vector, thus applying noise to the output of the generator. Ideally, this should allow generation of probability vectors whilst still providing samples to the discriminator. Unfortunately, our experiments with Gumbel-Softmax in section 3.4 showed significantly reduced performance over the added

input noise approach. We attribute this to the fact that the final Softmax used by Gumbel-Softmax still produces outputs which are often far from one-hot vectors, leading to many of the same problems which occur when trying to generate probabilities directly using WGAN.

$\ell 2^2$ WGAN

One possible solution to the issues in lemma 3.2.1 arises if we consider additional distance metrics. Equation 2.5 can be relaxed for an arbitrary distance metric as:

$$\begin{aligned} \mathcal{W}(\mathbb{P}(R), \mathbb{P}(Y)) = &\sup_f \mathbb{E}_{r\sim\mathbb{P}(R)}[f(y)] - \mathbb{E}_{y\sim\mathbb{P}(Y)}[f(x)], \\ &\text{subject to} f(R) - f(y) \leq c(y, r). \end{aligned} \tag{3.2}$$

where $c(y, r)$ is a cost function requiring that $c(x, x) = 0\ \forall x$.

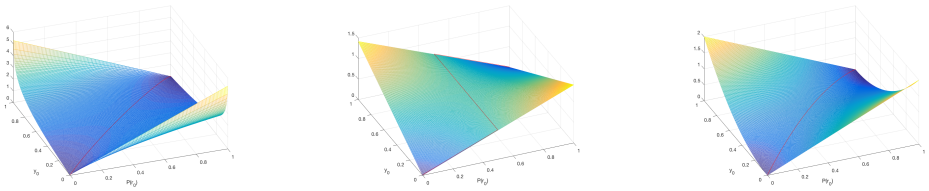
The EM distance can be written in this form by considering $c(y, r) = \|y - r\|$. An alternative would be to use $c(y, r) = \|y - r\|^2$, or the squared-$\ell 2$ distance. In fact, reworking the proof from Lemma 3.2.1 with this metric yields a gradient of 0 when $y = \mathbb{P}(R)$, suggesting that minimizing the Wasserstein-1 distance with this cost function would place y at the desired location.

However, the relaxed equation 3.2 no longer contains the 1-Lipschitz constraint on F which was present in equation 2.5. Thus, finding a minimum of equation 3.2 only constrains f at points which exist either in $\mathbb{P}(R)$ or $\mathbb{P}(Y)$. For disjoint distributions, the value of f between any pairs of points (y, r) is not defined and the path between them may be highly non-convex, making gradient decent from y to r impractical. This issue of disjoint distributions is shared with the original GAN formulation, and was discussed briefly in section 2.7. One solution is to add Gaussian noise to y and r, which makes the distributions overlap [18]. This added noise could be effective here as well, but has proven a difficult technique to use. Additionally, extending the use of squared-$\ell 2$ to the generation of sequences of probability vectors is non-trivial.

3.3 Architecture

This section describes the various GAN architectures we considered. We explored both RNN and CNN variants for generators and CNN variants for discriminators. We did not explore RNN discriminators because the majority of current frameworks do not support second order derivatives for RNN's, which are necessary for the regularization term in WGAN-GP (equation 2.7).

Enforcing 1-Lipschitz through regularization in equation 2.7 necessitates the ability to linearly interpolate between all pairs of points y and r. However, sequence of text vary in length, making



(a) Cross Entropy **(b)** EM (Wasserstein-1 $\ell 2$) **(c)** Wasserstein-1 $\ell 2^2$

Figure 3·2: Loss for length 2 probability vector. Real probability $\mathbb{P}(R = r_0)$ is shown on the x-axis. $\mathbb{P}(R = r_1) = 1 - \mathbb{P}(R = r_0)$. Network output x_0 is shown on the y-axis, with $y_1 = 1 - y_0$. Minimum of the loss is shown by the red lines. For both Cross Entropy and Wasserstein-1 $\ell 2^2$ the minimum loss occurs when $y_0 = \mathbb{P}(R = r_0)$. However, for EM distance the minimum loss occurs when y is 1-hot and follows the maximum value of r.

it impossible to interpret between arbitrary sentences. Thus applying WGAN-GP to text requires all sentences are either clipped or padded out to a common fixed length. Many previous GAN approaches do not take advantage of this fact [24, 25]. In the following sections we explore several different generator and discriminator architectures, many of which are made possible because of this fixed length.

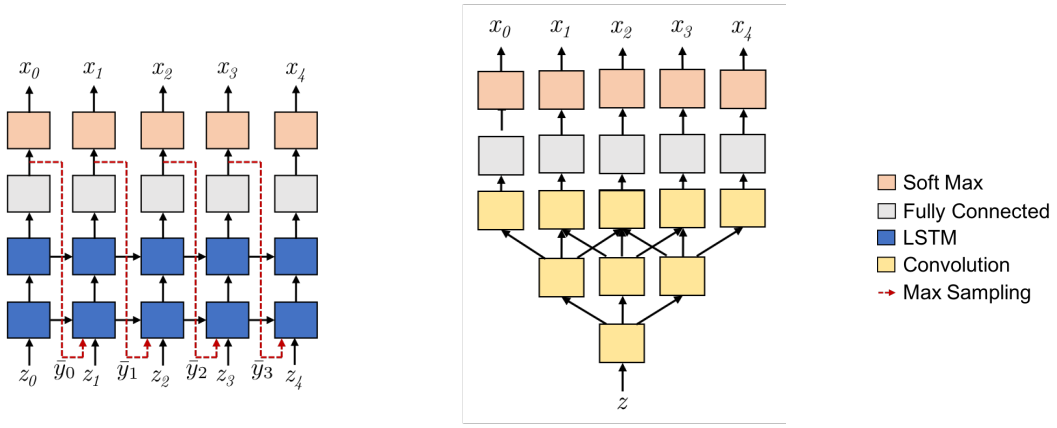
3.3.1 Generators

Generator architectures are shown in figure 3·3. Our RNN generator (figure 3·3a) is a close as possible to a traditional language model with the necessary modifications for added noise. The network consists of n Long Short-Term Memory (LSTM) layers [15] each with a hyperbolic tangent (Tanh) non-linearity. These layers are followed by two fully connected layers, with a Tanh non-linearity in between and Softmax at the output. For each time-step, the network takes as input a noise vector z, and the clamped previous output $\bar{r}_{t-1}$. Generator outputs, y_t are passed to the discriminator directly. However, y_t may not be strictly one-hot so we clamp it to the closest one hot vector before using it as input to the next time-step:

$$\bar{r}_t = \text{OneHot}(\text{Argmax}(y_t)).$$

Note that Argmax is non-differentiable, and we do not propagate gradients back through $\bar{r}_t$, which is consistent with standard neural language models.

We also tested a typical 1D CNN generator, shown in figure 3·3b, which is based roughly off the DCGAN architecture [21]. This architecture take as input a single noise vector z which is up-sampled to T using 1D Transpose Convolution layers with stride 2. This is followed by two fully



(a) RNN Generator

(b) Convolution-Transpose Generator

Figure 3·3: Generator architectures. (a): RNN (LSTM) based architecture, similar to a language model but with the addition of input Gaussian noise z. (b): Convolution based generator. 1-D version of DCGAN architecture which uses transpose-convolution operators for upsampling.

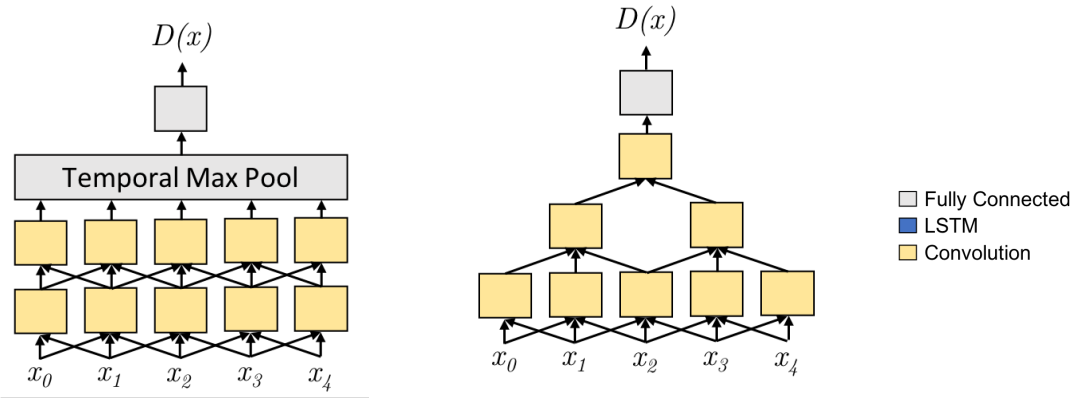
connected layers with Tanh non-linearity and Softmax output. The CNN generator does not include any feedback from one y_t to another, instead producing the entire y sequence in parallel.

3.3.2 Discriminators

Discriminator architectures are shown in figure 3·4. We explore a CNN discriminator architecture similar to the text classification network proposed by Zhang et al [34], shown in 3·4a. For this network, the input is passed to two convolution layers with a stride of 1. The output of these convolutional layers is then passed through a temporal max pooling which combines all time-steps into a single output vector. Finally, this output is run through a series of fully connected layers, the last of which has no non-linearity and only a single output neuron.

The temporal max-pooling in the text classification network is required to handle variable length sequences. However, since the WGAN-GP formulation requires fixed length discriminator input, this max pooling is no longer necessary and can be replaced with a concatenation along the time domain instead. Networks with this modification in our experiments are labeled as "Cat".

Fixed length inputs also allow us to explore DCGAN based discriminators, shown in figure 3·4b. These architectures are similar in many ways to the text classifications one, except for their use of convolutions with a stride of 2, which down-sample over the time dimension. When padding for these layers is chosen correctly they produces only a single time-step output from the last convolutional layer, removing the need for concatenation or max pooling before the output fully connected layers.



(a) Convolution Discriminator

(b) Convolution Downsampling Discriminator

Figure 3·4: Discriminator architectures. (a): Convolutional discriminator which uses a temporal max pooling layer to combine all time-steps. (b) Convolutional discriminator which uses a stride of 2 to downsample at each layer. Similar to a 1-D DCGAN style discriminator. (c) RNN based discriminator which reads in inputs in series.

3.3.3 Regularization

We attempt to improve stability of WGAN training by modifying regularization. Typically WGAN-GP regularization follows equation 2.7, with $\hat{y}$ sampled linearly between pairs of points y and r, given as:

$$\hat{y} = \epsilon * y + (1 - \epsilon) * r.$$

We vary this interpolation by sampling ϵ as the first output from a Dirichlet distribution with concentrations $(\alpha, 1.0)$. An $\alpha < 1.0$ causes ϵ to be concentrated closer to 1, and thus $\hat{y}$ closer to y, which puts additional emphasis on ensuring gradients close to generated data points are correct.

We also combat instability in training with a novel regularization term which penalizes the distance of $D(r)$ from 0:

$$\text{ZeroReg} = \delta * D(r)^2. \tag{3.3}$$

Note that if D is 1-Lipschitz the value $D(y)$ of generated points y are limited by the closest real value point r. In other words, $D(y) \leq \sup_{r \in \mathbb{P}(R=r)} D(r) + \|r - y\|$. When δ is large, $D(r)$ is forced to 0 for all r, which in turn means $D(y)$ is the distance between y and the closest r. This makes the gradient $\nabla_y D(y)$ point directly toward the closest r, creating a system which very stably moves all y's toward real valued points. However it also incurs mode dropping since this system accounts for only the distance between y and its closest r and does not match the distributions $\mathbb{P}(Y)$ and $\mathbb{P}(R)$.

Because of this, we found small values of δ to be practical choices, with best results occurring when δ is annealed to be greater than zero only toward the end of training. This effectively creates two stages of training. First $\delta = 0$ allows the generated distribution to spread out over the distribution $\mathbb{P}(R)$, with some instability. Then the regularization term is applied which clamps generated points to real valued points, producing realistic outputs.

3.3.4 embedding domain

Many neural language modeling systems have achieved improved performance through the use of pretrained embedding vectors, for example word-to-vec [35]. Embedding vectors are useful to cast from the high dimensional vocabulary space to an embedding space with a more manageable size. For our models we use a pretrained embedding vectors at the interface between the generator and discriminator. This is achieved by having the Generator produce outputs, y, in the embedded space directly, which can then be used as input to the discriminator. By having the generator output in the embedding space it can also take advantage of the pretrained embedding space, which puts similar words close together. Additionally, by outputting in the smaller embedding space we avoid handling the very large vocabulary space. Candidate generated vectors $\bar{r}_t$ are produced from the embedded outputs by locating the closest embedding vector to the produced output y_t.

3.4 Experiments

3.4.1 Context Free Grammar

We created a dataset for evaluation from a probabilistic context free grammar (PCFG). CFG's define a formal grammar with a group of one-to-many rules, called productions. PCFGs extend CFGs by adding a probability for each production. These probabilities allow us to increase the difficulty of the dataset by ensuring a high degree of variance across the probabilities of outputs.

The PCFG we use is included in Appendix 3.B. It contains 21 Non-Terminals and 39 Terminals. When unrolled so that each production is unique their are a total of 1174 productions. From the PCFG we randomly generate 1M training examples up to a max length of 25. Examples which are less than length 25 are zero padded out to 25. Random generation is accomplished by starting at the root node of the CFG and expanding in a depth-first manner until terminals are reached. The choice of which rule to expand at each production is made using the probabilities in the PCFG.

The PCFG defines all possible strings which can be generated by the grammar, which allows us to compute metrics which are unavailable on real data. We can fit generated output to the PCFG

to determine first if the output could be generated by the PCFG, and then the set of productions which generate it. The set of productions used for a given output, known as its parse-tree, allows us to take advantage of coverage metrics defined for trees. We compute the following metrics:

1. **Accuracy** (Acc) - the percentage of outputs of the GAN which are valid outputs of the CFG
2. **Production Coverage** (PC) - the percentage of productions in the unrolled PCFG which were reached using parse-trees for generated outputs.
3. **Edge Coverage** (EC) - the percentage of edges between productions in the unrolled PCFG which were reached using generated outputs. A values of 1.0 means the network generated all possible outputs of the PCFG.
4. **Sum Production Jensen-Shannon Divergence** (SPJS) - The sum of Jensen-Shannon divergences for all productions in the CFG. The Jensen-Shannon divergence for each production is computed by comparing the production probabilities in the CFG to estimated probabilities from the parse-trees of generated data. This metric is a proxy for the true Jensen-Shannon divergence between the generated and real distributions, since the true divergence is infinite unless all possible outputs are capable of being generated.

Note that we compute PC, EC and SPJS on the unrolled PCFG of 1174 productions. Also, because PC, EC, and SPJS require a parse tree they can only be computed using outputs which fit the CFG. Non valid outputs are used only to compute accuracy, and discarded before computation of PC, EC, and SPJS. As such, these metrics should be treated with care on results with low accuracy values.

We train our networks using the RMSPROP optimizer and batch size of 512. The discriminator is updated 10 times per generator update with learning rates of $5x10^{-4}$ and $2.5x10^{-4}$ for discriminator and generator respectively. Networks are trained for a total of 200 epochs. The first two epoch are used to initialize the discriminator, and the generator is not updated during this time. The generator learning rate is decreased by 0.4 at epochs 100, 130, 160, 175, and 190. Results for our experiments are shown in table 3.1. All experiments were run 10 times and then metrics were computed on 1M outputs from each run. We report the mean and standard deviation across the runs for the four metrics described above.

The first set of experiments compares the Added Noise, Gumbel Softmax, and Policy Gradient approaches. All three methods used an identical generator and discriminator network. For Gumbel Softmax we followed the methods of Jang et al. [32] as closely as possible. We annealed the

temperature parameter on the output Softmax as:

$$\max(0.5, e^{-rt}),$$

where t is the iteration and r is $1e-5$. For Policy Gradient we follow the methodology of Yu et al. [23]. The Added Noise method seems to perform significantly better than either the Gumbel Softmax or Policy Gradient methods. We attribute the stark difference in accuracy's to the limitations of the Gumbel Softmax and Policy Gradient approaches described in section 3.2

The second set of experiments compare the baseline model, of an RNN Generator and CNN discriminator, to other possible architectures. We draw several conclusions from this section. First, we found a significant drop in performance for CNN based generators (DC Gen) over RNN ones. The best performing CNN Generator is shown in the results table, but this performance drop was seen across the large array of CNN architectures we tried, varying kernel and number of hidden layers. This suggests that modeling text sequentially is important. Secondly, replacing the temporal max pooling with a concatenation operator (Cat) seems to increase performance on all metrics. We believe this is because the concatenation allows the discriminator to more easily achieve the 1-Lipschitz constraint than with max pooling. Thirdly, performance using a DC like Discriminator (DC Disc) is mixed, showing a decrease in Accuracy but an increase in SPJS and EC. Lastly, results using a ReLU on the second to last layer of the discriminator showed decreased performance. This is somewhat counterintuitive, since the linearity of the ReLU would seem to pair well with the 1-Lipschitz constraint.

The third set of experiments explores additional regularization techniques, applied to the baseline model. Increasing WGAN regularization, $lambda$, from the default 10 to 100 ($lambda100$) seems to have a positive effect on SPJS and EC, with a small impact on Accuracy. This effect is amplified when regularization is moved more heavily toward generated data points by setting α to 0.3 ($lambda100$ $\alpha 0.3$). This combination achieved the best Sum Jensen Shannon of all of our models. Conversely, added regularization on the distance of real data points from 0 ($\delta 1.0$) shows an increase in Accuracy at a cost of SPJS and EC. The loss of SPJS and EC was mitigated slightly by annealing delta over the course of training ($\delta 1.0$ ann), where delta is held at 0 until epoch 100 and then annealed as:

$$min(1.0, e^{c(epoch-100)}),$$

where c is chosen to be $log(0.1)/100$. This was improved further through the addition of concatena-

Table 3.1: PCFG results

Architecture	**Gen** Type	Hidden Size	Kernel	Stride	Act	FC Size	Act	**Disc** Type	Hidden Size	Kernel	Stride	Act	Inter Type	FC Size	Act	α	λ	δ	Acc	SPJS	PC	EC
Baselines																						
Added Noise	RNN	3x256	-	-	Tanh	1x256	Tanh	Conv	2x256	3	1	ReLU	Max Pool	1x256	Tanh	1.0	10	0.0	0.973 (0.009)	1.194 (0.465)	1.0 (0.0)	0.973 (0.02)
Policy Gradient																			0.892 (0.13)	8.72 (1.626)	0.998 (0.002)	0.943 (0.014)
Gumbel-Softmax																			0.677 (0.023)	18.93 (3.105)	0.993 (0.003)	0.917 (0.012)
Architecture Variations																						
Cat													Cat						0.987 (0.004)	0.671 (0.526)	1.0 (0.0)	0.994 (0.008)
DC Gen	Conv			2	ReLU														0.757 (0.013)	4.59 (0.708)	0.997 (0.006)	0.987 (0.008)
DC Disc									3x256		2		-	-	-				0.965 (0.014)	0.847 (0.462)	1.0 (0.0)	0.990 (0.012)
Disc ReLU															ReLU				0.863 (0.031)	3.686 (0.892)	1.0 (0.0)	0.932 (0.025)
DC Disc Tanh									3x256		2		-	-					0.967 (0.009)	0.892 (0.355)	1.0 (0.0)	0.988 (0.015)
DC Disc ReLU									3x256		2		-	-	ReLU				0.887 (0.008)	2.558 (1.415)	1.0 (0.0)	0.952 (0.031)
Gen No FC						-	-												0.975 (0.008)	1.067 (0.380)	1.0 (0.0)	0.974 (0.019)
Gen ReLU							ReLU												0.971 (0.014)	0.638 (0.079)	1.0 (0.0)	0.994 (0.003)
Regularization Variations																						
α0.3																0.3			0.984 (0.003)	1.214 (0.442)	1.0 (0.0)	0.965 (0.017)
λ100																	100		0.971 (0.002)	0.671 (0.525)	1.0 (0.0)	0.994 (0.008)
δ1.0																		1.0	0.992 (0.007)	4.89 (2.669)	0.996 (0.007)	0.949 (0.029)
δ1.0 ann																		0.0-1.0	0.996 (0.005)	4.56 (2.469)	0.996 (0.007)	0.951 (0.028)
α0.3 λ100																0.3	100		0.975 (0.004)	**0.419** (0.296)	1.0 (0.0)	**0.998** (0.001)
δ1.0 ann Cat													Cat					0.0-1.0	**0.997** (0.003)	0.658 (0.330)	1.0 (0.0)	0.987 (0.015)
δ1.0 ann α0.3 λ100																0.3	100	0.0-1.0	0.991 (0.004)	1.136 (0.5139)	1.0 (0.0)	0.980 (0.018)

tion (δ1.0 ann Cat), which achieves the best accuracy of any of our models.

3.4.2 One Billion Word

We test our networks on real world data from the 1-billion word dataset [26]. We followed a modern language modeling approach and model the data at the sub-word level using a BPE encoding with a vocab size of 50,000 [36]. The BPE is computed on the entirety of the 1-billion word dataset, and then used to segment the sentences into sub-words. We then train embedding vectors of size 512 on the sub-word encoded sentences using a skipgram model [35]. These embedding vectors are used in two ways. For both models they are used as an embedding for the inputs $\bar{r}$ in the generator. For the +EmbOut model, embeddings are also used at the interface between generator and discriminator as described in section 3.3.4. For GAN training we use the BPE-encoded sentences, but either clip or pad them out to a length of 10 words.

We trained generative models using an RNN generator and CNN discriminator with added input noise. The model is similar to the baseline model in the previous section, but we found that we needed to increase the size of the network to achieve reasonable results. We increased the size of all the generator layers to 1024 neurons and use a 3 layer LSTM. We also increased the discriminator layers to 512 and used 3 convolutional layers. We use the same training hyperparameters as the previous section, except we use a batch size of 1024 and only initialize the discriminator over the first epoch, since each epoch contains significantly more data than our CFG dataset. Generated examples are shown in table 3.2.

Our models are able to generate fairly realistic English sentences. Significantly more realistic results are generated when the generator and discriminator share the pretrained embedding domain. Without the embedding, the network struggles to handle the large $50,000$ word vocabulary, and to put words in the correct order.

Table 3.2: Example Outputs on One Billion Word Dataset. The Baseline model uses a RNN generator with added noise and CNN discriminator. The Baseline +EmbOut model adds that the generator outputs in the embedding domain.

Architecture	Examples
Baseline	Its they will be people from points at a , I has with already court smithsonian , instead to be For was too unacceptable , one of which the house This on the political , the , buchan was return
Baseline + EmbOut	Indeed you know the driver. Detainees said it was by the world hills . Mr. biden is left to reassemble the government powerhouse. Peterborough will get a third-largest rousing to make her patrols.

3.5 Conclusion

In this chapter we studied the production of discrete sentences using a GAN framework. We first proved that a WGAN system cannot be directly used to generate probability vectors as outputs, and then described several alternative approaches to avoid this issue. We then explored several potential architectures including additions to WGAN regularization and using a pretrained embedding to interface generator and discriminator. We tested these networks on data generated from a PCFG, which allows rigorous comparison between different network architectures. Finally, we applied our best architecture to the Google One Billion Word dataset and generated realistic looking English sentences using only a pretrained embedding.

Although this chapter makes some significant progress, GAN based training of discrete sequences is a difficult problem and is far from solved. We hope the techniques described in this chapter will inspire continued research in this area.

Appendix

3.A Proof of Lemma 3.2.1

Proof. Since y is a fixed probability vector, we can replace $\mathbb{E}_{y\sim\mathbb{P}(Y)}[f(y)]$ in 2.5 with $f(y)$. Equation 2.5 is then maximized if for every point $r \sim \mathbb{P}(R)$ the distance $f(r) - f(y)$ is maximized. Since f is bounded as 1-lipshitz this is simply the distance between points r and y or $\|r - y\|$. We can thus rewrite equation 2.5 as

$$\begin{aligned} W(\mathbb{P}(R), \mathbb{P}(Y)) &= \frac{\sum\limits_{r\sim\mathbb{P}(R)} \|r - y\|}{N} \\ &= \frac{\sum\limits_{i=0}^{N} \sum\limits_{r=v_i} \|r - y\|}{N} \\ &= \sum_{i=0}^{N} \mathbb{P}(R = v_i)\|v_i - y\| \end{aligned}$$

where v_i is the ith possible value of r. Here we are using the fact that each sample r must be drawn from a set of N discrete values.

Minimizing the Wasserstein-1 distance is thus equivalent to solving

$$\min_y \sum_{v\in V} \mathbb{P}(R = v)\|v - y\|,$$

where V is the set of all possible values of r, represented as one-hot vectors. V can also be thought of as points on the N-dimensional standard simplex, with y laying somewhere on the interior of that simplex.

Let v_{il} be the lth element of the ith vertex, or

$$v_{il} = \begin{cases} 1 \text{ if } i = l \\ 0 \text{ otherwise} \end{cases}$$

and β be an indicator variable defined as

$$\beta_{ij} = \begin{cases} -1 \text{ if } i = j \\ 1 \text{ otherwise} \end{cases}$$

We can compute the derivative of the kth element of y_k:

$$
\begin{aligned}
&\frac{\partial}{\partial y_k}\sum_{i=0}^{N}\mathbb{P}(R=v_i)\|v_i-y\| \\
&=\sum_{i=0}^{N}\mathbb{P}(R=v_i)\frac{1}{2\|v_i-y\|}\sum_{l}2(v_{il}-y_l)\frac{\partial}{\partial y_k}v_{il}-y_l \\
&=\sum_{i=0}^{N}\frac{\mathbb{P}(R=v_i)\sum\limits_{l}\beta_{kl}(v_{il}-y_l)}{\|v_i-y\|} \\
&=\sum_{i=0}^{N}\frac{\mathbb{P}(R=v_i)[\sum\limits_{l}\beta_{kl}(v_{il})-\sum\limits_{l}\beta_{kl}(y_l)]}{\|v_i-y\|} \\
&=\sum_{i=0}^{N}\frac{\mathbb{P}(R=v_i)[\beta_{ki}-(1-2y_k)]}{\|v_i-y\|}
\end{aligned}
$$

Letting $F(i)=\frac{\mathbb{P}(R=v_i)}{\|v_i-y\|}$

$$
\begin{aligned}
&=\sum_{i=0}^{N}F(i)[\beta_{ki}-(1-2y_k)] \\
&=\sum_{i=0}^{N}F(i)\beta_{ki}-\sum_{i=0}^{N}F(i)(1-2y_k) \\
&=\sum_{i=0}^{N}F(i)\beta_{ki}-\sum_{i=0}^{N}F(i)+\sum_{i=0}^{N}F(i)2y_k \\
&=-2F(k)+2y_k\sum_{i=0}^{N}F(i) \\
F(k)&=y_k\sum_{i=0}^{N}F(i) \\
\frac{\mathbb{P}(R=v_k)}{\|v_k-y\|}&=y_k\sum_{i=0}^{N}\frac{\mathbb{P}(R=v_i)}{\|v_i-y\|}
\end{aligned}
$$

Letting $y=\mathbb{P}(R)$ and thus $y_k=\mathbb{P}(R=v_k)$.

$$
\begin{aligned}
\frac{y_k}{\|v_k-y\|}&=y_k\sum_{i=0}^{N}\frac{y_i}{\|v_i-y\|} \\
\frac{1}{\|v_k-y\|}&=\sum_{i=0}^{N}\frac{y_i}{\|v_i-y\|} \\
0&=\sum_{i=0}^{N}\frac{y_i-v_{ik}}{\|v_i-y\|}
\end{aligned}
$$

This holds iff $y_k = 0$ or $y_k = 1$ or $y_i = y_j \ \forall i, j$. Which means y, and by extension $\mathbb{P}(R)$ must be one-hot or must generate all values with equal probability. In all other cases, the gradient will not be zero, which concludes the proof.

□

3.B Probabilistic Context Free Grammar

PCFG used for 3.4.1. Non terminals are represented using capital letters and terminals in quotes. Each production consists of the starting non-terminal on the left followed by lists of terminals and non terminals separated by "—" on the right. Probabilities for each rule are shown in square brackets.

```
PCFG =
  S → SOS PHRASE EOS [1.0]
  SOS → '1' [1.0]
  EOS → '2' [1.0]
  PHRASE → NP VPP [0.8] | NP VPP CONJADV NP VPP [0.2]
  NP → Det Nom [0.7] | PropN [0.3]
  Nom → Adj N [0.4] | N [0.6]
  VPP → VP [0.5] | VP PRO VP [0.3] | VP CONJP [0.2]
  PP → P NP [1.0]
  PropN → '3' [0.4] | '4' [0.5] | '5' [0.1]
  Det → '6' [0.6] | '7' [0.4]
  N → '8' [0.1] | '9' [0.1] | '10' [0.2] | '11' [0.2] | '12' [0.4]
  Adj → '13' [0.25] | '14' [0.2] | '15' [0.4] | '16' [0.1] | '17' [0.05]
  V → '18' [0.2] | '19' [0.2] | '20' [0.1] | '21' [0.1] | '22' [0.4]
  VPT → '23' [0.3] | '24' [0.3] | '25' [0.1] | '26' [0.05] | '27' [0.25]
  PRO → '28' [0.75] | '29' [0.25]
  CONJP → CONJ CV VPT NPRO [1.0]
  CV → '30' [0.9] | '31' [0.1]
  CONJ → '32' [0.5] | '33' [0.5]
  P → '34' [0.2] | '35' [0.8]
  NPRO → '36' [0.4] | '37' [0.6]
  CONJADV → '38' [0.9] | '39' [0.1]
```

Chapter 4

Learning to Repair Software Vulnerabilities with Generative Adversarial Networks

4.1 Introduction

As discussed in chapter 1, one of the main issues preventing the adoption of NMT techniques for many applications is a lack of training data. We address this problem by adopting an approach that removes the need for paired data by using Generative Adversarial Networks (GANs) [17]. We keep a traditional NMT model as the generator, and replace the typical negative likelihood loss with an adversarial discriminator, building on the methods discussed in chapter 3. This discriminator is trained to distinguish between generated outputs and real examples of desired output, and so it serves as a proxy for the discrepancy between the generated and real distributions.

This approach has three main difficulties. We discussed in chapter 3 that sampling from the output of traditional NMT systems, in order to produce discrete outputs, is non-differentiable. As a solution we follow the Added Noise methodology discussed in chapter 3, by adding an input noise vector to the decoder of the NMT system which allow it to directly estimate samples from the target distribution. Secondly, adversarial training does not guarantee that the generated output will correspond to the input (i.e. the generator is trained to match distributions, not samples). To enforce the generator to generate useful outputs we condition our generator on the input by incorporating two novel generator loss functions. Thirdly, the domains we consider are not bijective, i.e., for repair tasks a bad sample can have more than one repair, and conversely a good one can be broken in more than one way. This motivates our choice of conditional loss functions, since previous loss functions do not handle this limitation. We should note that although the experiments in this chapter are on source code, the approach and the techniques proposed in this chapter are application-agnostic in that they can be applied to other similar problems, such as correcting grammar errors or converting between negative and positive sentiments (e.g., in online reviews.). Additionally, while software vulnerability repair is a harder problem than detection, our proposed repair technique can

leverage the same datasets used for detection and yields a much more explainable and useful tool than detection alone.

4.2 Related Work

4.2.1 Software Repair

Much research has been done on automatic repair of software. Here we describe previous data-driven approaches (see [37] for a more extensive review of the subject). Two successful recent approaches are that of Le et al. [38] and Long and Rinard [39]. Le et al. mine a history of bug fixes across multiple projects and attempt to reuse common bug fix patterns on newly discovered bugs. Long and Rinard learn and use a probabilistic model to rank potential fixes for defective code. These works, along with the majority of past work in this area, require a set of test cases which is used to rank and validate produced repairs. Devlin et al. [40] avoid the need for test cases by generating repairs with a rule based method and then ranking them using a neural network. Gupta et al. [41] take this one step further by training a NMT model to directly generate repairs for incorrect code. Hence, the work in [41] most closely resembles our work, but has the major drawback of requiring paired training data.

4.2.2 Sequence Generation using GANs

As discussed in chapter 3, one of the main issues in applying GANs to NMT tasks is the handling of discrete data (e.g. text). Typically, the output of a NMT network uses a softmax to produce a probability vector over the set possible outputs. This probability vector is then sampled to produce an output token. This sampling operation is non-differentiable and therefore prevents us from using the gradients of the discriminator to train the generator. A few approaches have had success despite this difficulty. One such approach is that of Yu et al. [23], which treats the output of the discriminator as a reward in a reinforcement learning setting. This allows the sampling of outputs from the generator since gradients do not need to be passed back through the discriminator. However, since a reward is provided for the entire sequence, gradients computed for the generator do not provide information on which parts of the output sequence the discriminator thinks is incorrect, resulting in long convergence times. Several other approaches have had success with directly applying an adversarial discriminator to the output of a sequence generator without sampling. Both Press et al. [25] and Rajeswar et al. [24] are able to generate fairly realistic looking sentences of modest length in this way using Wasserstein GAN [19].

Another issue we must consider is that the output of the generator should be conditioned upon the input. This is implicit with NMT since we have pairs of data, but is not enforced with the adversarial loss in a GAN. Work has been done on how to condition a GAN's generator on an input sequence $\mathbf{x}$ instead of a random variable. This can easily be performed when paired data is available, by providing the discriminator with both x and $G(x)/y$, thereby formulating the problem as in the conditional approach of Mirza and Osindero [42, 43]. However, this is clearly more difficult when pairs are not available. One approach is to enforce conditionality through the use of dual generators which translate between domains in opposite directions. For example, Gomez et al. apply the cycle GAN [44] approach to cipher cracking [28]. They train two generators, one to take raw text and produce ciphered text, and the other to undo the cipher. Having two generators allows Gomez et al. to encrypt raw data using the first generator, then decrypt it with the other, ensuring conditionality by adding a loss function which compares this doubly translated output with the original raw input. Lample et al. [27] adopt a somewhat similar approach for Neural Machine Translation. They translate using two encoder/decoder pairs which convert from a given language to a latent representation and back respectively. They then use an adversarial loss to ensure that the latent representations are the same between both languages, thus allowing translation by encoding from one language and then decoding into the second. For conditionality they adopt a similar approach to Gomez et al. by fully translating a sentence from one language to another, translating it back, and then comparing the original sentence to the produced double translation.

The approaches of both Gomez et al. and Lample et al. rely on the ability to transform a sentence across domains in both directions. This makes sense in many translation spaces as there are a finite number of reasonable ways to transform a sentence in one language to a correct one in the other. This allows for a network which finds a single mapping from every point in one domain to a single point in the other domain, to still cover the majority of translations. Unfortunately, in a sequence correction task, one domain contains all correct sequences, while the other contains everything not in the correct domain. Therefore, the mapping from correct to incorrect is not one-to-one, it is one to many. A single mapping discovered by a network would fail to elaborate the space of all bad functions, thus enforcing conditionality only on the relatively small set of bad functions it covers. As such we propose to enforce conditionality using a self-regularization term on the generator, similar in nature to that used by Shrivastava et al. [45] to generate realistic looking images from simulated ones.

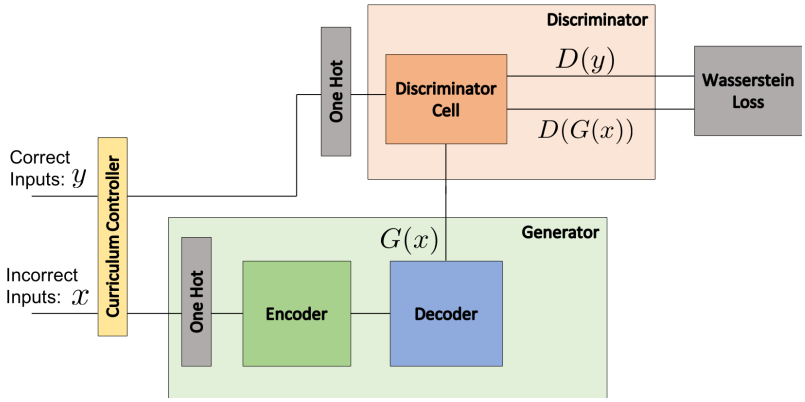


Figure 4·1: Block diagram of GAN approach.

4.3 Approach

Our approach aims to combine a traditional NMT model with a GAN for training. As such our generator consists of a standard LSTM based NMT system with an attention mechanism similar to that of Luong et al [7]. Our discriminator uses a convolution based approach, since they have been shown to be easier to train and to generally perform better than RNN based discriminators [43]. The output of our generator is used directly as the input to the discriminator (no sampling). A block diagram of the approach is show in figure 4·1.

4.3.1 Domain Mapping with Self-Regularization

In the context of sequence correction, we need to constrain our generated samples $G(x)$ to be a corrected versions of x. Therefore, we have the following two requirements: (1) correct sequences should remain unchanged when passed through the generator; and (2) repaired sequences should be close to the original corresponding incorrect input sequences.

We explore two regularizers to address these requirements. As our first regularizer we train our generator as an autoencoder on data sampled from correct sequences. This directly enforces item (1), while indirectly enforcing item (2) since the autoencoder loss encourages subsequences which are correct to remain unchanged. The autoencoder regularizer is given as

$$\mathcal{L}_{AUTO}(G) = \mathbb{E}_{r \sim P(r)} \left[-r \log(G(r)). \right] \tag{4.1}$$

As our second regularizer, we enforce that the frequency of each token in the generated output remains close to the frequency of the input tokens. This enforces item (2) with the exception that it may allow changes in the order of the sequence, e.g., arbitrary reordering does not increase this loss.

However, the GAN loss alleviates this issue since arbitrary reordering produces incorrect sequences which differ heavily from $p(r)$. Our second regularizer is given as

$$\mathcal{L}_{FREQ}(G) = \mathbb{E}_{x\sim P(x)}[\ \sum_{i=0}^{n} \|\text{freq}(x,i) - \text{freq}(G(x),i)\|_2^2\ ]. \tag{4.2}$$

where n is the size of the vocabulary and $\text{freq}(x,i)$ is the frequency of the i^{th} token in x.

4.3.2 Putting It All Together - Proposed GAN Framework

The generator in our network consists of a standard NMT system with an attention mechanism similar to that of Luong et al [7]. For all experiments the encoder and decoder consist of multi-layer RNNs utilizing Long Short-Term Memory (LSTM) units [15]. We use a dot-product attention mechanism as per [7]. We use convolution based discriminators since they have been shown to be easier to train and to generally perform better than RNN based discriminators [43]. Additional network details are provided in Appendix 4.A

We have two different regularized loss models given as:

$$\mathcal{L}(D,G) = \mathcal{L}_{WGAN}(D,G) + \lambda\mathcal{L}_{AUTO}(G) \tag{4.3}$$

$$\mathcal{L}(D,G) = \mathcal{L}_{WGAN}(D,G) + \lambda\mathcal{L}_{FREQ}(G) \tag{4.4}$$

where $\mathcal{L}_{AUTO}(G)$ and $\mathcal{L}_{FREQ}(G)$ are defined in Section 4.3.1. We also experiment with the unregularized base loss model where we set $\lambda = 0$.

4.3.3 Autoencoder Pre-Training

We rely heavily on pre-training to give our GAN a good starting point. Our generators are pre-trained as de-noising autoencoders on the desired data [46]. Specifically we train the generator with the loss function:

$$\mathcal{L}_{AUTO\_PRE}(G) = \mathbb{E}_{r\sim P(r)}\ [-r\log(G(\hat{r})),] \tag{4.5}$$

where $\hat{r}$ is a noisy version of the input created by dropping tokens in r with small probability (eg. 0.2) and randomly inserting and deleting n tokens, where n is a small fraction sequence length (eg. 0.03).

4.3.4 Curriculum Learning

Likelihood based methods for training NMT systems often utilize teacher forcing during training where the input to the decoder is forced to be the desired value regardless of what was generated at the previous time step [16]. This allows stable training of very long sequence lengths even at the start of training. Our adversarial method cannot use teacher forcing since the desired sequence is unknown, and must therefore always pass a sample of the previous output as the next input. This can lead to unstable training, since errors early in the output will be propagated forward, potentially creating gibberish in the latter parts of the sequence. To avoid this problem we adopt a curriculum learning strategy where we incrementally increase the length of produced sequences throughout training. Instead of selecting subsets of the data for curriculum training, we clip all sequences to have a predefined maximum length for each curriculum step. Although this approach relies on the discriminator being able to handle incomplete sentences, it did not degrade the performance as long as the discriminator was briefly retrained after each curriculum update.

4.4 Experiments

GAN methods have often been criticized for their lack of easy evaluation metrics. Therefore, we focus our experiments on datasets which contain paired examples. This enables us to meaningfully evaluate the performance of our approach, even though our GAN approach does not require pairs to train. These datasets also allow us to train NMT networks and use their performance as an upper bound to our GAN based approach. We start our experiments by exploring two hand-curated datasets, namely sequences of sorted numbers and Context Free Grammar (CFG), which help highlight the benefits of our proposed GAN approach to address the domain mapping problem. We then investigate the harder problem of correcting errors in C/C++ code. All of our results are given in Table 4.1.

4.4.1 Sorting

In order to show the necessity of enforcing accurate domain mapping we generate a dataset where the repair task is to sort the input into ascending order. We generate sequences of 20 randomly selected integers (without replacement) between 0 and 50 in ascending order. We then inject errors by swapping n selected tokens which are next to each other, where n is a (rounded) Gaussian random variable with mean 8 and standard deviation 4. The task is then to sort the sequence back into its original ascending order given the error injected sequence. This scheme of data generation allows us to maintain pairs of good (before error injection) and bad (after error injection) data, and to

compute the accuracy with which our GAN is able to restore the good sequences from the bad. We refer to this accuracy as 'Sequence Accuracy' (or Seq. Acc.). In order to assess our domain mapping approach and evaluate the usefulness of our self-regularizer loss functions defined in Section 4.3.1, we also compute the percentage of sequences which have valid orderings but not necessarily valid domain mappings, which we refer to as 'Order Accuracy' (or Order Acc.).

It is clear from the results in Table 4.1 that the vanilla (base) GAN easily learns to generate sequences with valid ordering, without necessarily paying attention to the input sequence. This leads to high Order Accuracy, but low Sequence Accuracy. However, adding Auto or Freq loss regularizers, as in (4.3) and (4.4), significantly improves the Seq. Acc., which shows that these losses do effectively enforce correct mapping between source and target domains.

4.4.2 Simple Grammar

For our second experiment, we generate data from a simple Context Free Grammar similar to that used by Rajeswar et al. [24]. The specifics of the CFG is provided in Appendix 4.B. Our good data is selected randomly from the set of all sequences which satisfy the grammar and are less than length 20. We then inject errors into each sequence, where the number of errors is chosen as a Gaussian random variable (zero thresholded and rounded) with mean 5 and standard deviation 2. Each error is then randomly chosen to be either a deletion of a random token, insertion of a random token, or swap of two random tokens.

The network is tasked with generating the original sequence from the error injected one. This task better models real data than the sorting task above, because each generated token must follow the grammar and is therefore conditioned on all previous tokens. The results in Table 4.1 show that our proposed GAN approach is able to achieve high CFG accuracy, in terms of generating correct sequences that fit the CFG. In addition to CFG accuracy, we also compute BLEU scores based on the pairs before and after error injection. We should note that our random error injection process results in many bad examples corresponding to a specific good example or vice verse, i.e., mappings are not bijective. Having multiple bad examples in the dataset paired with the same good example contributes to the slightly lower BLEU scores, since the network can only map each bad input to a single output. This issue appears frequently in real world repair datasets, since code sequences can be repaired or broken multiple different ways. Our GAN approach performs well on this CFG dataset suggesting that it can handle this issue for which cycle approaches are not appropriate [28, 27, 44].

4.4.3 SATE IV

SATE IV is a dataset which contains C/C++ synthetic code examples (functions) with vulnerabilities from 116 different Common Weakness Enumeration (CWE) classes, and was originally designed to explore performance of static and dynamic analyzers [47]. Each bad function contains a specific vulnerability, and is paired with several candidate repairs. There is a total of $117,738$ functions of which $41,171$ contain a vulnerability and $76,567$ do not. We lex each function using our custom lexer. After lexing, each function ranges in length from 10 to 300 tokens.

Using this data, we created two datasets to perform two different experiments, namely *paired* and *unpaired* datasets. The paired dataset allows us to compare the performance of our GAN approach with a NMT approach. In order to have a dataset which is fair for both GAN and NMT training, we created paired data by taking each example of vulnerable code and sampling one of its repairs randomly. We iterate this process through the dataset four times, pairing each vulnerable function with a sampled repair, and combine the resulting sets into a single large dataset. We should mention that although the paired dataset includes labeled pairs, those labels are not utilized for GAN training. For the unpaired dataset, we wanted to guarantee that a given source sequence does not have a corresponding target sequence anywhere in the training data. To achieve this, we divided the data into two disjoint sets by placing either a vulnerable function or its candidate repairs into the training dataset with equal probability. Note that this operation reduces the size of our training data by half. For testing, we compute BLEU scores using all of the candidate repairs for each vulnerable function. We use a 80/10/10% train/validation/test split.

As shown in Table 4.1, our proposed GAN approach achieves progressively better results when we add (a) curriculum training, and (b) either $\mathcal{L}_{AUTO}$ or $\mathcal{L}_{FREQ}$ regularization loss. The Base + Cur + Freq model proves to be the best among different GAN models, and performs reasonably close to the NMT baseline, which is the upper performance bound. The results on the unpaired dataset are fairly close to those achieved by the paired dataset, particularly in the Base case, even though they are obtained with only half of the training data. Some code examples where our GAN makes correct repairs are provided in Table 4.2, with additional examples Appendix 4.C.

4.5 Conclusions

We have proposed a GAN based approach to train an NMT system for discrete domain mapping applications. The major advantage of our approach is that it can be used in the absence of paired data, opening up a wide set of previously unusable data sets for the sequence correction task. Key

Table 4.1: Results on all experiments. Cur refers to experiments using curriculum learning, while Auto and Freq are those using $\mathcal{L}_{AUTO}$ and $\mathcal{L}_{FREQ}$, respectively. Sate4-P and Sate4-U denote paired and unpaired datasets, respectively.

	Sorting		CFG		Sate4-P	Sate4-U
Model	Seq Acc.	Order Acc	BLEU-4	CFG Acc	BLEU-4	BLEU-4
NMT						
Base	**99.7**	**99.8**	**91.3**	**99.3**	96.3	N/A
Base + Cur	**99.7**	**99.8**	90.2	98.9	**96.4**	N/A
Proposed GAN						
Base	82.8	96.9	88.5	98.0	84.2	79.3
Base + Auto	98.9	99.6	**88.6**	96.5	85.7	79.2
Base + Freq	**99.3**	**99.7**	88.3	97.5	86.2	79.5
Base + Cur	81.5	98.0	88.4	**98.9**	88.3	81.1
Base + Cur + Auto	96.2	98.0	88.5	97.8	89.9	**81.5**
Base + Cur + Freq	98.2	99.1	**88.6**	96.3	**90.3**	81.3

Table 4.2: Successful Repairs: (Top) This function calls sprintf to print out two strings, but only provides the first string to print. Our GAN repairs it by providing a second string. (Bottom) This function uses a variable again after freeing it. Our GAN repairs it by removing the first free.

With Vulnerability	Repaired
`void CWE685_Function_Call_With_Incorrect_ Number_Of_Arguments() { char dst[DST_SZ]; sprintf(dst, "%s %s", SRC_STR); printLine(dst); }`	`void CWE685_Function_Call_With_Incorrect_ Number_Of_Arguments() { char dst[DST_SZ]; sprintf(dst, "%s %s", SRC_STR, SRC_STR); printLine(dst); }`
`void CWE415_Double_Free__malloc_free_ struct_31() { twoints *data; data = NULL; data = (twoints *)malloc(100 * sizeof(twoints)); free(data); { twoints *data_copy = data; twoints *data = data_copy; free(data); } }`	`void CWE415_Double_Free__malloc_free_ struct_31() { twoints *data; data = NULL; data = (twoints *)malloc(100 * sizeof(twoints)); { twoints *data_copy = data; twoints *data = data_copy; free(data); } }`

to our approach is the addition of two novel generator loss functions which enforce accurate domain mapping without needing multiple generators or domains to be bijective. We also have discussed, and made some progress, toward handling discrete outputs with GANs. We note that this problem is far from solved, however, and will likely inspire more research. Even though we only apply our approach to the problem of source code correction, it is applicable to other sequence correction problems, such as Grammatical Error Correction or language sentiment translation, e.g., converting negative reviews into positive ones.

Appendix

4.A Network and Training Details

Here we provide additional network and training details useful for experimental replication. All of the networks used in this chapter use a similar architecture but vary in the number and size of layers.

4.A.1 Network Architecture

For all the experiments, we use identical networks for the Generator in our GAN model as well as in the NMT model in our seq2seq baseline. Thus when we refer to generator in the rest of the section it applies to both models. Our network architecture is shown in Figure 4·2.

Our generator consists of two RNNs, an encoder and a decoder. For Sorting and CFG experiments, the generator RNNs contain 3 layers of 512 neurons each. For Sate4, it contains 4 layers of 512 neurons each. The encoder RNN processes the input sequence and produces a set of hidden states h_t. The final hidden state h_T is used as the initial state to the decoder RNN which generates outputs s_t one at a time, feeding its outputs back as input to t_1 until an end of sequence character is produced. The decoder and encoder are linked using global dot product attention as per [7].

All networks share the same discriminator architecture. Discriminator inputs $(s_0, s_1 ... s_T)$ each in $\mathbb{R}^k$ are concatenated into a matrix $\mathbf{G}$ in $\mathbb{R}^{T \times k}$. They are then passed through a single 1D convolutional layer with 300 filters each of sizes of 3, 7, and 11. These outputs are then aggregated and fed into a max pooling operation over the entire sequence length. This is fed into two fully connected layers, the first with 512 neurons, and the second with a single neuron, the output of the discriminator.

4.A.2 Training

We first train our generator as a denoising autoencoder for which we use the Adam optimization algorithm with a learning rate of 10^{-4}. The same pretrained network is used to initialize the generator for all GAN and seq2seq networks.

GAN networks are trained using the RMSprop optimization algorithm. Learning rates are initialized to $5 * 10^{-4}$ for the discriminator and 10^{-5} for the generator. We train the discriminator

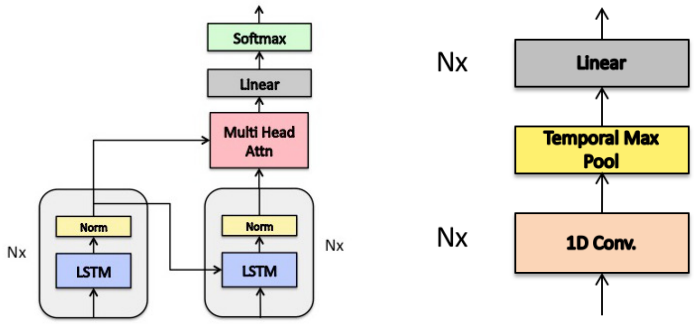


Figure 4·2: (Left) Generator consisting of N encoder layers feeding N decoder layers. Outputs from the encoder are also used as inputs to the attention mechanism with the query coming from the decoder output. (Right) Discriminator consisting of N convolution layers, a temporal max pooling, and N fully connected layers.

15 times for every generator update. Seq2seq models are trained using the Adam optimizer with a learning rate of 10^{-4}. We experimented extensively with varying the learning rate but found that increasing the discriminator learning rate made it unstable causing its accuracy to decrease. Increasing the generator learning rate causes it to update to quickly for the discriminator, meaning the discriminator would not remain close to optimal and therefore gradients through it were not reliable. In order to ensure that the discriminator starts at a good initial point, we initialize it by training it alone for the first 10 epochs. The generators learning rate is decayed by a factor of 0.9 every 10 epochs. In models where we employ curriculum learning, this decay is only performed after the curriculum is completed. Networks are trained for 200, 400, and 1000 epochs for the sorting, CFG, and SATE4 experiments, respectively.

Our curriculum clips each sequence to a given length. We step up the curriculum length either when the discriminator accuracy falls below 55% or after 40 epochs, whichever comes first. Sorting and CFG curriculum starts at sequence length 5 and is increased by 2 at each step. SATE4 curriculum starts at length 75 and is increased by 5 at each step.

4.B Context Free Grammar

Our simple CFG experiment uses the following CFG. Each line is a production rule with possible sequences separated by |. Symbols in quotes are terminals.

```
S: SOS NP VP EOS
SOS: '1'
EOS: '2'
NP: Det Nom | PropN
Nom: Adj N | N
VP: V NP | V NP PP
PP: P NP
PropN: '3' | '4' | '5'
Det: '6' | '7'
N: '8' | '9' | '10' | '11' | '12'
Adj : '13' | '14' | '15' | '16' | '17'
V: '18' | '19' | '20' | '21'
P: '22' | '23'
```

4.C Repair Examples

Here we provide additional selected examples of source code correctly and incorrectly repaired by our GAN model. Tables 4.3-4.6 show successfull repairs, and Tables 4.7-4.8 show failures.

Table 4.3: Successful Repair - This functions reads the index of an array access from a socket and returns the memory at the index. The vulnerable function only checks the lower bound on the array size. Our GAN repairs it by adding an additional check on the upper bound.

With Vulnerability

```
void CWE129_Improper_Validation_Of_
      Array_Index() {
  int data;
  data = -1;
  {
    ifdef _WIN32 WSADATA wsa_data;
    int wsa_data_init = 0;
    endif int recv_rv;
    struct sockaddr_in s_in;
    SOCKET connect_socket = INVALID_SOCKET;
    char input_buf[CHAR_ARRAY_SIZE];
    do {
      ifdef _WIN32 if (WSAStartup(MAKEWORD(2, 2),
            &wsa_data) != NO_ERROR) break;
      wsa_data_init = 1;
      endif connect_socket = socket(AF_INET, SOCK_STREAM,
            IPPROTO_TCP);
      if (connect_socket == INVALID_SOCKET)
        break;
      memset(&s_in, 0, sizeof(s_in));
      s_in.sin_family = AF_INET;
      s_in.sin_addr.s_addr = inet_addr("127.0.0.1");
      s_in.sin_port = htons(TCP_PORT);
      if (connect(connect_socket, (struct sockaddr *)\&s_in,
            sizeof(s_in)) ==
          SOCKET_ERROR)
        break;
      recv_rv = recv(connect_socket, input_buf,
            CHAR_ARRAY_SIZE, 0);
      if (recv_rv == SOCKET_ERROR || recv_rv == 0)
        break;
      data = atoi(input_buf);
    } while (0);
    if (connect_socket != INVALID_SOCKET)
      CLOSE_SOCKET(connect_socket);
    ifdef _WIN32 if (wsa_data_init) WSACleanup();
    endif
  }
  {
    int data_copy = data;
    int data = data_copy;
    {
      int data_buf[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
      if (data >= 0) {
        printIntLine(data_buf[data]);
      } else {
        printLine("ERROR: Array index is negative");
      }
    }
  }
}
```

Repaired

```
void CWE129_Improper_Validation_Of_
      Array_Index() {
  int data;
  data = -1;
  {
    ifdef _WIN32 WSADATA wsa_data;
    int wsa_data_init = 0;
    endif int recv_rv;
    struct sockaddr_in s_in;
    SOCKET connect_socket = INVALID_SOCKET;
    char input_buf[CHAR_ARRAY_SIZE];
    do {
      ifdef _WIN32 if (WSAStartup(MAKEWORD(2, 2),
            &wsa_data) != NO_ERROR) break;
      wsa_data_init = 1;
      endif connect_socket = socket(AF_INET, SOCK_STREAM,
            IPPROTO_TCP);
      if (connect_socket == INVALID_SOCKET)
        break;
      memset(&s_in, 0, sizeof(s_in));
      s_in.sin_family = AF_INET;
      s_in.sin_addr.s_addr = inet_addr("127.0.0.1");
      s_in.sin_port = htons(TCP_PORT);
      if (connect(connect_socket, (struct sockaddr *)&s_in,
            sizeof(s_in)) ==
          SOCKET_ERROR)
        break;
      recv_rv = recv(connect_socket, input_buf,
            CHAR_ARRAY_SIZE, 0);
      if (recv_rv == SOCKET_ERROR || recv_rv == 0)
        break;
      data = atoi(input_buf);
    } while (0);
    if (connect_socket != INVALID_SOCKET)
      CLOSE_SOCKET(connect_socket);
    ifdef _WIN32 if (wsa_data_init) WSACleanup();
    endif
  }
  {
    int data_copy = data;
    int data = data_copy;
    {
      int data_buf[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
      if (data >= 0 && data < 10) {
        printIntLine(data_buf[data]);
      } else {
        printLine("ERROR: Array index is out-of-bounds");
      }
    }
  }
}
```

Table 4.4: Successful Repair - This function attempts to accept a socket and use it before it has bound it. Our GAN approach repairs the function by reordering the bind, listen, and accept into the correct order.

With Vulnerability

```
void CWE666_Operation_on_Resource_in_Wrong_
      Phase_of_Lifetime__accept_listen_bind_() {
  {
    char data[100] = "";
    ifdef _WIN32 WSADATA wsa_data;
    int wsa_data_init = 0;
    endif int recv_rv;
    struct sockaddr_in s_in;
    char *replace;
    SOCKET listen_socket = INVALID_SOCKET;
    SOCKET accept_socket = INVALID_SOCKET;
    size_t data_len = strlen(data);
    do {
      ifdef _WIN32 if (WSAStartup(MAKEWORD(2, 2),
            &wsa_data) != NO_ERROR) break;
      wsa_data_init = 1;
      endif listen_socket = socket(AF_INET, SOCK_STREAM,
            IPPROTO_TCP);
      if (listen_socket == INVALID_SOCKET)
        break;
      memset(&s_in, 0, sizeof(s_in));
      s_in.sin_family = AF_INET;
      s_in.sin_addr.s_addr = INADDR_ANY;
      s_in.sin_port = htons(TCP_PORT);
      accept_socket = accept(listen_socket, NULL,
            NULL);
      if (accept_socket == SOCKET_ERROR)
        break;
      if (listen(listen\_socket, LISTEN\_BACKLOG)
          == SOCKET\_ERROR)
        break;
      if (bind(listen_socket, (struct sockaddr *)& s_in,
          sizeof(s_in)) == SOCKET_ERROR)
        break;
      recv_rv = recv(accept_socket, (char *)data + data_len,
                (int)(100 - data_len - 1), 0);
      if (recv_rv == SOCKET_ERROR || recv_rv == 0)
        break;
      data[recv_rv] = '\0';
      replace = strchr(data, '\r');
      if (replace)
        *replace = '\0';
      replace = strchr(data, '\n');
      if (replace)
        *replace = '\0';
    } while (0);
    if (listen_socket != INVALID_SOCKET)
      CLOSE_SOCKET(listen_socket);
  }
}
```

Repaired

```
void CWE666_Operation_on_Resource_in_Wrong_
      Phase_of_Lifetime__accept_listen_bind_() {
  {
    char data[100] = "";
    ifdef _WIN32 WSADATA wsa_data;
    int wsa_data_init = 0;
    endif int recv_rv;
    struct sockaddr_in s_in;
    char *replace;
    SOCKET listen_socket = INVALID_SOCKET;
    SOCKET accept_socket = INVALID_SOCKET;
    size_t data_len = strlen(data);
    do {
      ifdef _WIN32 if (WSAStartup(MAKEWORD(2, 2),
            &wsa_data) != NO_ERROR) break;
      wsa_data_init = 1;
      endif listen_socket = socket(AF_INET, SOCK_STREAM,
            IPPROTO_TCP);
      if (listen_socket == INVALID_SOCKET)
        break;
      memset(&s_in, 0, sizeof(s_in));
      s_in.sin_family = AF_INET;
      s_in.sin_addr.s_addr = INADDR_ANY;
      s_in.sin_port = htons(TCP_PORT);
      if (bind(listen_socket, (struct sockaddr*)&s_in,
          sizeof(s_in)) == SOCKET_ERROR)
        break;
      if (listen(listen_socket, LISTEN_BACKLOG)
          == SOCKET\_ERROR)
        break;
      accept_socket = accept(listen_socket, NULL,
            NULL);
      if (accept_socket == SOCKET_ERROR)
        break;
      recv_rv = recv(accept_socket, (char *)data + data_len,
                (int)(100 - data_len - 1), 0);
      if (recv_rv == SOCKET_ERROR || recv_rv == 0)
        break;
      data[recv_rv] = '\0';
      replace = strchr(data, '\r');
      if (replace)
        *replace = '\0';
      replace = strchr(data, '\n');
      if (replace)
        *replace = '\0';
    } while (0);
    if (listen_socket != INVALID_SOCKET)
      CLOSE_SOCKET(listen_socket);
  }
}
```

Table 4.5: Successful Repair - This function has a buffer allocated which is too small for the resulting data write. Our GAN repairs it by increasing the amount of memory allocated to the buffer.

With Vulnerability

```
void CWE131_Incorrect_Calculation_Of_
      Buffer_Size() {
  wchar_t *data;
  data = NULL;
  data = (wchar_t *)malloc(10 * sizeof(wchar_t));
  {
    wchar_t data_src[10 + 1] = SRC_STRING;
    size_t i, src_len;
    src_len = wcslen(data_src);
    for (i = 0; i < src_len; i++) {
      data[i] = data_src[i];
    }
    data[wcslen(data_src)] = L '\0';
    printWLine(data);
    free(data);
  }
}
```

Repaired

```
void CWE131_Incorrect_Calculation_Of_
      Buffer_Size() {
  wchar_t *data;
  data = NULL;
  data = (wchar_t *)malloc((10 + 1) * sizeof(wchar_t));
  {
    wchar_t data_src[10 + 1] = SRC_STRING;
    size_t i, src_len;
    src_len = wcslen(data_src);
    for (i = 0; i < src_len; i++) {
      data[i] = data_src[i];
    }
    data[wcslen(data_src)] = L '\0';
    printWLine(data);
    free(data);
  }
}
```

Table 4.6: Successful Repair - This function calls sprintf to print out two strings, but only provides the first string to print. Our GAN repairs it by providing a second string.

With Vulnerability	Repaired
`void CWE685_Function_Call_With_Incorrect_ Number_Of_Arguments() { char dst[DST_SZ]; sprintf(dst, "%s %s", SRC_STR); printLine(dst); }`	`void CWE685_Function_Call_With_Incorrect_ Number_Of_Arguments() { char dst[DST_SZ]; sprintf(dst, "%s %s", SRC_STR, SRC_STR); printLine(dst); }`

Table 4.7: Incorrect Repair - This function tries to print out from data which has not been fully initialized. Our GAN corrects this by fully initializing the variable, but then attempts to call a variable identifier instead of making the appropriate function call.

With Vulnerability	GAN Attempted Repaired	Golden Repair
`void CWE457_Use_of _Uninitialized_Variable() { double *data; data = (double *)ALLOCA(10 * sizeof(double)); { int i; for (i = 0; i < (10 / 2); i++) { data[i] = (double)i; } } { int i; for (i = 0; i < 10; i++) { printDoubleLine(data[i]); } } }`	`void CWE457_Use_of_ Uninitialized_Variable(){ double *data; data = (double *)ALLOCA(10 * sizeof(double)); { size_t i; for (i = 0; i < 10; i++) { data[i] = (double)i; } } { int i; for (i = 0; i < 10; i++) { static_five(data[i]); } } }`	`void CWE457_Use_of_ Uninitialized_Variable() { double *data; data = (double *)ALLOCA(10 * sizeof(double)); { int i; for (i = 0; i < 10; i++) { data[i] = (double)i; } } { int i; for (i = 0; i < 10; i++) { printDoubleLine(data[i]); } } }`

Table 4.8: Incorrect Repair - This function encounters an integer underflow by assigning a random value to a char and then subtracting 1. If the random value is 0 this will underflow the char. The given golden repair in this case is simply to change the random char to a known value. However, our GAN gets confused. It instead modifies the rand function in an unknown way and proceeds to free the data rather then print it.

With Vulnerability	GAN Attempted Repaired	Golden Repair
`void CWE191_Integer_ Underflow() { unsigned char data; data = ' '; data = (unsigned char)rand(); { unsigned char result = data - 1; printHexUnsignedCharLine(result); } }`	`void CWE191_Integer_ Underflow(){ unsigned char data; data = ' '; data = (unsigned char) rand((unsigned int)data); { char data = data; free(data); } }`	`void CWE191_Integer_ Underflow() { unsigned char data; data = ' '; data = 5; { unsigned char result = data - 1; printHexUnsignedCharLine(result); } }`

Chapter 5

Tree-Transformer: A Transformer-Based Method for Correction of Tree-Structured Data

5.1 Introduction

Most machine learning approaches to correction tasks operate on sequential representations of input and output data. Generally this is done as a matter of convenience — sequential data is readily available and requires minimal effort to be used as training data for many machine learning models. As discussed in the previous chapters, sequence-based machine learning models have produced prominent results in both translation and correction of natural language [6, 5, 4, 2, 1, 3, 48, 13].

Many sequential data types can also be more informatively represented using a tree structure. One common method to obtain tree-structured data is to fit sequential data to a grammar, such as a Context Free Grammar (CFG). The use of a grammar ensures that generated trees encode the higher-order syntactic structure of the data in addition to the information contained by sequential data. Examples of tree-structured data from text and code are show in figure 5·1.

In this chapter, we trained a neural network to operate directly on trees, teaching it to learn the syntax of the underlying grammar and to leverage this syntax to produce outputs which are grammatically correct. Our model, the Tree-Transformer, handles correction tasks in a tree-based encoder and decoder framework. The Tree-Transformer leverages the popular Transformer architecture [5], modifying it to incorporate the tree structure of the data by adding a parent-sibling tree convolution block. To show the power of our model, we focused our experiments on two common data types and their respective tree representations: Abstract Syntax Trees (ASTs) for code and Constituency Parse Trees (CPTs) for natural language.

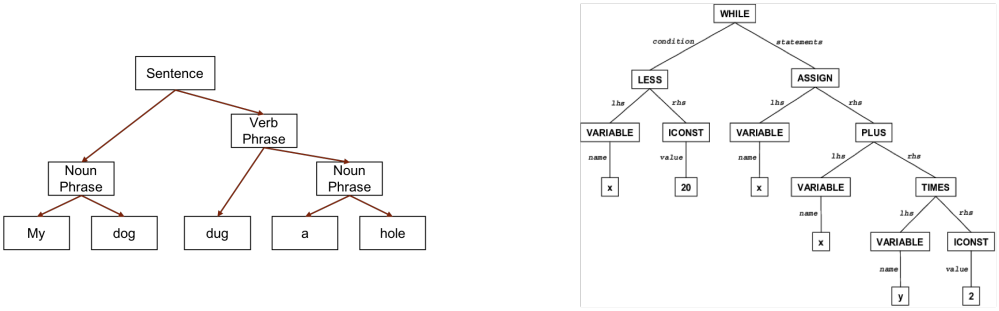


Figure 5·1: Examples of tree-structured data. Left: constituency parse tree for English text. Right: abstract syntax tree for code.

5.2 Related Work

5.2.1 Tree Structured Neural Networks

Existing work on tree-structured neural networks can largely be grouped into two categories: encoding trees and generating trees. Several types of tree encoders exist [49, 50, 51, 52]. The seminal work of Tai et al. [49] laid the ground work for these methods, using a variant of a Long Short Term Memory (LSTM) network to encode an arbitrary tree. A large body of work has also focused on how to generate trees [53, 54, 55, 56, 57, 58, 59, 60]. The work of Dong et al., [53] and Alvarez-Melis and Jaakkola [54] each extend the LSTM decoder popular in Neuro Machine Translation (NMT) systems to arbitrary trees. This is done by labeling some outputs as parent nodes and then forking off additional sequence generations to create their children. Only a small amount of work has combined encoding and generation of trees into a tree-to-tree system [61, 62]. Of note is Chakraborty et al. [62] who use a LSTM-based tree-to-tree method for source code completion.

To our knowledge our work is the first to use a Transformer-based network on trees, and apply Tree-to-Tree techniques to natural language and code correction tasks.

5.2.2 Code Correction

Code correction was discussed in chapter 4, with related work in section 4.2.1. Some information is repeated here for completeness.

There is extensive existing work on automatic repair of software. However, the majority of this work is rule-based systems which make use of small datasets (see [37] for a more extensive review of these methods). Two successful, recent approaches in this category are that of Le et al., [38] and Long and Rinard [39]. Le et al. mine a history of bug fixes across multiple projects and attempt

to reuse common bug fix patterns on newly discovered bugs. Long and Rinard learn and use a probabilistic model to rank potential fixes for defective code. Unfortunately, the small datasets used in these works are not suitable for training a large neural network like ours.

Neural network-based approaches for code correction are less common. Devlin et al. [40] generate repairs with a rule-based method and then rank them using a neural network. Gupta et al. [41] were the first to train a NMT model to directly generate repairs for incorrect code.

5.2.3 Grammatical Error Correction

Grammatical Error Correction (GEC) is the task of correcting grammatically incorrect sentences. This task is similar in many ways to machine translation tasks. However, initial attempts to apply NMT systems to GEC were outperformed by phrase-based or hybrid systems [63, 64, 65].

Initial, purely neural systems for GEC largely copied NMT systems. Yuan and Brisco [2] produced the first NMT style system for GEC by using the popular attention method of Bahdanau et al. [6]. Xie et al. [4] trained a novel character-based model with attention. Ji et al. [1] proposed a hybrid character-word level model, using a nested character level attention model to handle rare words. Schmaltz et al. [3] used a word-level bidirectional LSTM network. Chollampatt and Ng [48] created a convolution-based encoder and decoder network which was the first to beat state of the art phrased-based systems. Finally, Junczys-Dowmunt et al. [13] treated GEC as a low resource machine translation task, utilizing the combination of a large monolingual language model and a specifically designed correction loss function.

5.3 Architecture

Our Tree-Transformer architecture is based on the Transformer architecture of Vaswani et al. [5], modified to handle tree-structured data. The transformer architecture is described in section 2.6. Our major change to the Transformer is the replacement of the feed forward sublayer in both the encoder and decoder with a Tree Convolution Block (TCB). The TCB allows each node direct access to its parent and left sibling, thus allowing the network to understand the tree structure. We follow the same overall architecture for the Transformer as Vaswani et al., consisting of self-attention, encoder-decoder attention, and TCB sub layers in each layer. Our models follow the 6 layer architecture of the base Transformer model with sublayer outputs, d_{model}, of size 512 and tree convolution layers, d_{ff}, of size 2048.

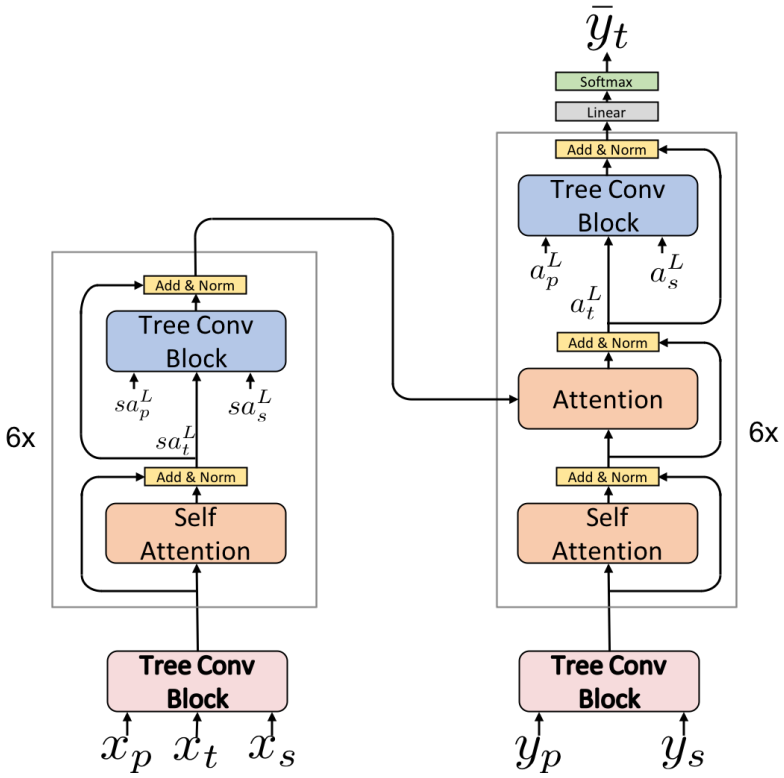


Figure 5·2: Tree-Transformer model architecture.

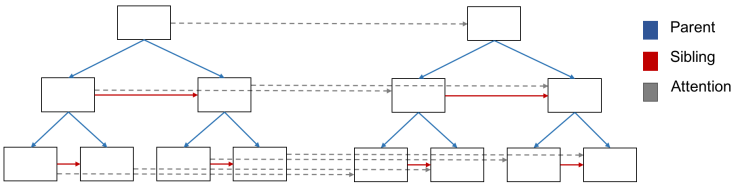


Figure 5·3: Tree-Transformer State Transfer

5.3.1 Parent-Sibling Tree Convolution

Tree convolution is computed for each node as:

$$TCB(x_t, x_p, x_s) = relu(x_t W_t + x_p W_p + x_s W_s + b)W_2 + b_2.$$

The inputs x_t, x_p, and x_s all come from the previous sublayer, x_t from the same node, x_p from the parent node, and x_s from its left sibling. In cases where the node does not have either a parent (e.g. the root node) or a left sibling (e.g. parents first child), the inputs x_p and x_s are replaced with a learned vector v_p and v_s respectively.

In addition to the TCB used in each sub layer, we also use a TCB at the input to both the encoder and decoder. In the encoder this input block combines the embeddings from the parent, the sibling and the current node (p, s, and t). In the decoder, the current node is unknown since it has not yet been produced. Therefore, this block only combines the parent and sibling embeddings, leaving out the input x_t from the equation above.

The overall structure of the network is shown in Figure 5·2. The inputs to each TCB come from the network inputs, x/y, for the input blocks, and from the previous sublayer, sa^L/a^L, for all other blocks.

5.3.2 Top-Down Encoder and Decoder

Both our encoder and decoder use a top-down approach where information flows from the tree's root node down toward to the leaf nodes, as shown in Figure 5·3. Thus, leaf nodes have access to a large sub-tree, while parent nodes have a more limited view. An alternative approach would be to use a bottom-up encoder, where each node has access to its children, and a top-down decoder which can disseminate this information. This bottom-up/top-down model is intuitive because information flows up the tree in the encoder and then back down the decoder. However, we found that utilizing the same top-down ordering for both encoder and decoder performed better, likely because the symmetry

in encoder and decoder allows decoder nodes to easily attend to their corresponding nodes in the encoder. This symmetry trivializes copying from encoder to decoder, which is particularly useful in correction tasks where large portions of the trees remain unchanged between input and output.

5.3.3 Generating Tree Structure

In order to generate each tree's structure, we treat each set of siblings as a sequence. For each set of siblings, we generate each node one at a time, ending with the generation of an end-of-sequence token. The vocabulary used defines a set of leaf and parent nodes. When a parent node is generated, we begin creation of that nodes children as another set of siblings.

5.3.4 Depth First Ordering

As with any NMT system, each output value y_t is produced from the decoder and fed back to subsequent nodes as input during evaluation. Ensuring inputs y_p and y_s are available to each node requires that parents are produced before children and that siblings are produced in left-to-right order. To enforce this constraint, we order the nodes in a depth-first manner. This ordering is shown by the numbering on nodes in Figure 5·4. The self attention mechanism in the decoder is also masked according to this order, so that each node only has access to previously produced ones.

5.3.5 No Positional Encoding

The Transformer architecture utilizes a positional encoding to help localization of the attention mechanisms. Positional encoding is not as important in our model because the TCB allows nodes to easily locate its parent and siblings in the tree. In fact, we found that inclusion of a Positional Encoding caused the network to overfit, likely due to the relatively small size of the correction datasets we use. Given this, our Tree-Transformer networks do not include a Positional Encoding.

5.4 Why Tree Transformer

In this section we motivate the design of our Tree-Transformer model over other possible tree-based architectures. Our choice to build upon the Transformer model was two-fold. First, Transformer-based models have significantly reduced time-complexity relative to Recurrent Neural Network (RNN) based approaches. Second, many of the building blocks required for a tree-to-tree translation system, including self-attention, are already present in the Transformer architecture.

5.4.1 Recurrent vs Attention Tree Networks

Many previous works on tree-structured networks used RNN-based tree architectures where nodes in layer L are given access to their parent or children in the same layer [53, 54, 62]. This state transfer requires an ordering to the nodes during training where earlier nodes in the same layer must be computed prior to later ones. This ordering requirement leads to poor time complexity for tree-structured RNN's, since each node in a tree needs access to multiple prior nodes (e.g. parent and sibling). Accessing the states of prior nodes thus requires a gather operation over all past produced nodes. These gather operations are slow, and performing them serially for each node in the tree can be prohibitively expensive.

An alternative to the RNN type architecture is a convolutional or attention-based one, where nodes in layer L are given access to prior nodes in layer $L-1$. With the dependence in the same layer removed, the gather operation can be batched over all nodes in a tree, resulting in one large gather operation instead of T. From our experiments, this batching resulted in a reduction in training time of two orders of magnitude on our largest dataset; from around 2 months to less than a day.

5.4.2 Conditional Probabilities and Self Attention

The Tree-Transformer's structure helps the network produce grammatically correct outputs. However, for translation/correction tasks we must additionally ensure that each output, y_t, is conditionally dependent on both the input, x, and on previous outputs, $y_{<t}$. Conditioning the output on the input is achieved using an encoder-decoder attention mechanism [6]. Conditioning each output on previous outputs is more difficult with a tree-based system. In a tree-based model like ours, with only parent and sibling connections, the leaf nodes in one branch do not have access to leaf nodes in other branches. This leads to potentially undesired conditional independence between branches. Consider the example constituency parse trees shown in Figure 5·4. Given the initial noun phrase "My dog", the following verb phrase "dug a hole" is far more likely than "gave a speech". However, in a tree-based model the verb phrases do not have direct access to the sampled noun phrase, meaning both possible sentences would be considered roughly equally probable by the model.

We address the above limitation with the inclusion of a self-attention mechanism which allows nodes access to all previously produced nodes. This mechanism, along with the depth-first ordering of the node described in section 5.3.4, gives each leaf node access to all past produced leaf nodes.

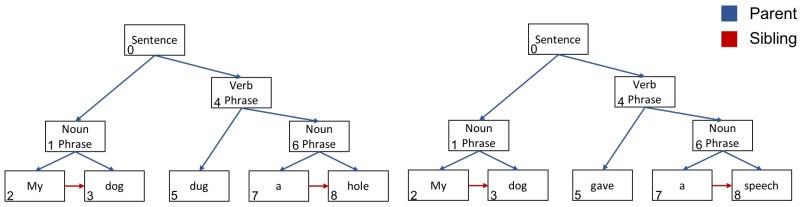


Figure 5·4: Example Constituency Parse Tree. The index of the node in depth-first ordering is shown in the bottom left of each node. Note: leaf nodes in the verb phrase do not have access to leaf nodes in the left noun phrase without self-attention

Our model fits the standard probabilistic language model of equation 2.2:

$$\mathbb{P}(R = r|x) = \prod_{t=1}^{T} p(r_t|r_{<t}, x),$$

where t is the index of the node in the depth first ordering.

5.5 Training

This section describes the training procedure and parameter choices used in our model. We trained our models in parallel on 4 Nvidia Tesla V-100 GPU's. Trees were batched together based on size with each batch containing a maximum $20,000$ words. We used the ADAM optimizer with inverse square root decay and a warm up of $16,000$ steps. A full list of hyperparameters for each run is included in the supplementary material.

5.5.1 Regularization

The correction datasets we used in this chapter are relatively small compared to typical NMT datasets. As such we found a high degree of regularization was necessary. We included dropout of 0.3 before the residual of each sub layer and attention dropout of 0.1. We also added dropout of 0.3 to each TCB after the non-linearity. We applied dropout to words in both source and target embeddings as per [13] with probabilities 0.2 and 0.1 respectively. We included label smoothing with $\epsilon_{ls} = 0.1$

5.5.2 Beam-Search

Because of the depth-first ordering of nodes in our model, we can use beam search in the same way as traditional NMT systems. Following equation 2.2, we can compute the probability of a generated sub-tree of t nodes simply as the product of probabilities for each node. We utilize beam-search

during testing with a beam width of 6. Larger beam widths did not produce improved results.

5.6 Experiments/Results

5.6.1 Code Correction

We trained our Tree-Transformer on code examples taken from the NIST SATE IV dataset [47]. SATE IV contains around $120K$ C and C++ files from 116 different Common Weakness Enumerations (CWEs), and was originally designed to test static analyzers. Each file contains a bad function with a known security vulnerability and at least one good function which fixes the vulnerability. We generate Abstract Syntax Trees (ASTs) from these functions using the Clang AST framework [66, 67].

To provide a representation which is usable to our network, we tokenize the AST over a fixed vocabulary in three ways. First, high level AST nodes and data types are represented by individual tokens. Second, character and numeric literals are represented by a sequence of ASCII characters with a parent node defining the kind of literal (e.g. Int Literal, Float Literal). Finally, we use variable renaming to assign per function unique tokens to each variable and string. Our vocabulary consists of 403 tokens made up of 60 AST tokens, 23 data type tokens, 256 ASCII tokens, and 64 Variable tokens.

Using the SATE IV dataset requires pre-processing, which we do during data generation. First, many of the SATE IV functions contain large amounts of dead code. In these cases, the bad and good functions contain largely the same code, but one path will be executed during the bad function and another in the good one. To make these cases more realistic, we removed the dead code. Second, although each file for a particular CWE contains unique functions at the text level, many of them are identical once converted to AST with renamed variables. Identical cases comes in two flavors: one where a bad function is identical to its good counterpart, and one where multiple bad functions from different files are identical. The first occurs commonly in SATE IV in cases where the bad and good functions are identical except for differing function calls. Since we operate at the function level, examples of this case are not useful and are removed. The second case occurs when bad functions differ only in variable names, strings, or function calls. To handle this, we compare all bad functions tree representations and combine identical bad functions into a single bad tree and a combination of all good trees from its component functions, with duplicate good trees removed. After pre-processing the data, we retain a total of $32,316$ bad functions and $47,346$ good functions. These are split 80/10/10 into training/validation/testing.

Table 5.1: SATE IV Results

Architecture	Precision	Recall	F0.5
4-layer LSTM	51.3	53.4	51.7
Transformer	59.6	86.1	63.5
Tree-Transformer	**84.5**	**89.4**	**85.4**

To our knowledge this processing of the SATE IV dataset is new. As such, we compare our network to two NMT systems operating on sequence-based representation of the data; a 4 Layer LSTM with attention and a base Transformer model. These sequence-based models use a representation with an almost identical vocabulary and tokenization to our tree representation but they operate over the tokenized sequence output of the Clang Lexer instead of the AST.

During testing, we utilized clangs source-to-source compilation framework to return the tree output of our networks to source code. We then compute precision, recall, and F0.5 scores of source code edits using the MaxMatch algorithm [68]. Results are given in Table 5.1. Our Tree-Transformer model performs better than either of the other two models we considered. We believe this is because source code is more naturally structured as a tree than as a sequence, lending itself to our tree-based model.

5.6.2 Grammar Error Correction

We applied our tree-based model as an alternative to NMT and phrase-based methods for GEC. Specifically, we encoded incorrect sentences using their constituency parse trees, and then generated corrected parse trees. Constituency parse trees represent sentences based on their syntactic structure by fitting them to a phrase structured grammar. Words from the input sentence become leaf nodes of their respective parse trees, and these nodes are combined into phrases of more complexity as we progress up the tree [69, 51]

A large amount of research has focused on the generation of constituency parse trees [70, 71, 72]. We utilize the Stanford NLP group's shift-reduce constituency parser [73] to generate trees for both incorrect and correct sentences in our datasets. These are represented to our network with a combined vocab consisting of $50K$ word level tokens and 27 parent tokens. The parent tokens come from the part of speech tags originally defined by the Penn Treebank [74] plus a root token. Following recent work in GEC [13, 48], the word level tokens are converted into sub-words using a Byte Pair Encoding (BPE) trained on the large Wikipedia dataset [75]. The BPE segments rare words into multiple subwords, avoiding the post-processing of unknown words used in many existing GEC techniques.

We test our network on two GEC benchmarks: the commonly used NUCLE CoNLL 2014 task [76], and the AESW dataset [77]. CoNLL 2014 training data contains $57K$ sentences extracted from essays written by non-native English learners. Following the majority of existing GEC work, we augment the small CoNLL dataset with the larger NIST Lang-8 corpus. The Lang-8 data contains $1.1M$ sentences of crowd sourced data taken from the Lang-8 website, making it noisy and of lower quality than the CoNLL data. We test on the $1,312$ sentences in CoNLL 2014 testing set and use the $1,381$ sentences of the CoNLL 2013 testing set as validation data. For evaluation we use the official CoNLL M2scorer algorithm to determine edits and compute precision and recall [68].

We also explore the large AESW dataset. AESW was designed in order to train grammar error identification systems. However, it includes both incorrect and corrected versions of sentences, making it useful for GEC as well. AESW contains $1.2M$ training, $143K$ testing, and $147K$ validation sentences. The AESW data was taken from scientific papers authored by non-native speakers, and as such contains far more formal language than CoNLL.

GEC Training

For GEC we include a few additions to the training procedure described in Section 5.5. First, we pre-train the network in two ways on $100M$ sentences from the large monolingual dataset provided by [63]. We pre-train the decoder in our network as a language model, including all layers in the decoder except the encoder-decoder attention mechanism. We also pre-train the entire model as a denoising-autoencoder, using a source embedding word dropout of 0.4.

For loss we use the edit-weighted MLE objective defined by Junczys-Dowmunt et al. [13]:

$$MLE(x, y) = -\sum_{t=1}^{T} \lambda(y_t) log P(y_t | x, y < t),$$

where (x, y) are a training pair, and $\lambda(y_t)$ is 3 if y_t is part of an edit and 1 otherwise. We compute which tokens are part of an edit using the python Apted graph matching library [78, 79, 80].

During beam-search we ensemble our networks with the monolingual language-model used for pre-training as per [4]:

$$s(y|x) = \sum_{t=1}^{T} log\mathbb{P}(Y_t = y_t | x, y_{<t}) + \alpha log\mathbb{P}_{lm}(Y_t = y_t | y_{<t}),$$

where α is chosen between 0 and 1 based on the validation set. Typically, we found an alpha of 0.15 performed best. Networks with $\alpha > 0$ are labeled with +Mon-Ens

Table 5.2: CoNLL 2014 results

Architecture	Precision	Recall	F0.5
Prior State-of-the-art Approaches			
Chollampatt and Ng 2017.	62.74	32.96	53.14
Junczys-Dowmunt and Grundkiewicz. 2016	61.27	27,98	49.49
Prior Neural Approaches			
Ji et al. 2017	-	-	45.15
Schmaltz et al. 2017	-	-	41.37
Xie et al. 2016	49.24	23.77	40.56
Yuan and Briscoe. 2016	-	-	39.90
Chollampatt and Ng. 2018	**65.49**	33.14	54.79
Junczys-Dowmunt et al. 2018	63.0	38.9	**56.1**
This Work			
Tree-Transformer	57.39	28.12	47.50
Tree-Transformer +Mon	58.45	30.42	49.35
Tree-Transformer +Mon +Mon-Ens	57.84	33.26	50.39
Tree-Transformer +Auto	65.22	30.38	53.05
Tree-Transformer +Auto +Mon-Ens	59.14	**43.23**	55.09

Table 5.3: CoNLL 2014 Example Output

Input	In conclusion , we could tell the benefits of telling genetic risk to the carriers relatives overweights the costs .
Labels	In conclusion , we **can see that** the benefits of telling genetic risk to the carrier**'s** relatives **outweighs** the costs . In conclusion , we **can see that** the benefits of **disclosing** genetic risk to the carriers relatives **outweigh** the costs. In conclusion , we **can see that** the benefits of **revealing** genetic risk to the carrier**'s** relatives **outweigh** the costs .
Network	In conclusion , **it can be argued** that the benefits of **revealing** genetic risk **to one 's** relatives **outweighs** the costs .

CoNLL 2014 Analysis

Results for CoNLL 2014 are provided in Table 5.2. Our Tree-Transformer achieves significantly higher recall than existing approaches, meaning we successfully repair more of the grammar errors. However, our precision is also lower which implies we make additional unnecessary edits. We attribute this drop to the fact that our method tends to generate examples which fit a structured grammar. Thus sentences with uncommon grammar tend to be converted to a more common way of saying things. An example of this effect is provided in table 5.3.

AESW Analysis

Results for AESW are provided in Table 5.4. We achieve the highest to date F0.5 score on AESW, including beating out our own sequence-based Transformer model. We attribute this to the fact that AESW is composed of samples taken from submitted papers. The more formal language used in this context may be a better fit for the structured grammar used by our model.

5.7 Conclusion

In this chapter we introduced the Tree-Transformer architecture for tree-to-tree correction tasks. We applied our method to correction datasets for both code and natural language and showed an increase

Table 5.4: AESW results

Architecture	Precision	Recall	F0.5
Prior Approaches			
Schmaltz et al. 2017 (Phrased-based)	-	-	38.31
Schmaltz et al. 2017 (Word LSTM)	-	-	42.78
Schmaltz et al 2017 (Char LSTM)	-	-	46.72
This Work			
Transformer (Seq to Seq)	52.3	36.2	48.03
Tree-Transformer	55.4	37.1	**50.43**

in performance over existing sequence-based methods. We believe our model achieves its success by taking advantage of the strong grammatical structure inherent in tree-structured representations. For the future we hope to apply our approach to other tree-to-tree tasks, such as natural language translation. Additionally, we intend to extend our approach into a more general graph-to-graph method.

Appendix

5.A hyperparameters

Hyperparameters utilized are listed in tables 5.5 and 5.6. Default hyperparameters are listed at the top of each table. A blank means the run utilized the default hyperparameter. An explanation of hyperparemeters follows.

- **N** - number of layers
- d_{model} - size of sublayer outputs - see [5]
- d_{ff} - size of inner layer in TDB/FF for Tree-Transformer/Transfomer
- h - Number of attention heads
- d_k - Size of keys in attention mechanism
- d_v - Size of values in attention mechanism
- p_{drop} - Dropout probability between sub-layers
- p_{dattn} - Dropout probability on attention mechanism - see [5]
- p_{dff} - Dropout probability on inner layer of TDB/FF for Tree-Transformer/Transfomer
- p_{des} - Source embedding word dropout probability
- p_{det} - Target embedding word dropout probability
- ϵ_{ls} - Label Smoothing ϵ
- **lr** - Learning Rate, We use isr learning rate as per [5]. As such this learning rate will never be fully met, maximum learning rate depends upon warmup.
- **warmup** - number of steps for linearly LR warmup
- **train steps** - total number of steps for training

Table 5.5: Model Parameters

Architecture	N	d_{model}	d_{ff}	h	d_k	d_v	p_{drop}	p_{dattn}	p_{dff}	p_{des}	p_{det}	ϵ_{ls}
default	6	512	2048	8	64	64	0.3	0.1	0.3	0.2	0.1	0.1
SATE IV												
LSTM Transformer Tree-Transformer	4	1024	N/a									
GEC Pretraining												
Monolingual Autoencoder										0.4		
Conll 2014												
Tree-Transformer Tree-Transformer +Mon Tree-Transformer +Mon +Mon-Ens Tree-Transformer +Auto Tree-Transformer +Auto +Mon-Ens												
Aesw												
Transformer Tree-Transformer												

Table 5.6: Training Parameters

Architecture	lr	warmup	train steps	Mon	Auto	Mon-Ens	EW-MLE	Time (Hours)
default	$d_{model}^{-0.5}$	4000	100k	-	-	-	-	
SATE IV								
LSTM Transformer Tree-Transformer								40 18 22
GEC Pretraining								
Monolingual Autoencoder			500k 500k					38 50
Conll 2014								
Tree-Transformer		16000					3	26
Tree-Transformer +Mon		16000		✓			3	26
Tree-Transformer +Mon +Mon-Ens		16000		✓		✓	3	26
Tree-Transformer +Auto		16000			✓		3	26
Tree-Transformer +Auto +Mon-Ens		16000			✓	✓	3	26
Aesw								
Transformer		16000		✓		✓	3	19
Tree-Transformer		16000		✓		✓	3	25

- **Mon** - Initialized from monolingual pre-trained network
- **Auto** - Initialized from autoencoder pre-trained network
- **Mon-Ens** - Ensemble trained network with monolingual netword during beam search as per [4]
- **EW-MLE** - Use edit-weight MLE objective function as per [13]
- **Time (Hours)** - Total training time

Chapter 6

Conclusion

This chapter provides concluding remarks on the previous three chapters, including discussion on contributions, strengths, weaknesses, and future work.

6.1 Summary and Contributions

- Chapter 3 presented methodology for training a language model with a WGAN. We first illustrated difficulties with directly training a language model with a WGAN and proved that we cannot train a language model to generate probabilities in this way. We then explored three alternative solutions which allow training of a language model with a WGAN. We presented an analysis of these alternatives on a dataset generated from a context free grammar, which showed that the added noise method performed the best of the three. We also explored both RNN and CNN generator architectures, several CNN based discriminator architectures, and two novel regularization techniques. Finally, we used the best performing network on the context free grammar dataset, plus a novel use of pertained embeddings, to train on the one billion word dataset, producing realistic English sentences.

- In Chapter 4 we designed a system which uses a WGAN and conditional losses to train a NMT system on unpaired data. We tested two different losses to enforce conditionality between the input and output of the NMT system, one which uses an autoencoder loss on real data, and one which penalizes differences in the frequency of tokens between input and output. Additionally, we used autoencoder pre-training to initialized the network and during training used curriculum learning to incrementally step up sequence lengths. We trained our network to correct vulnerabilities in code using the SATE IV datasets, and achieved results close to those obtainable using paired data.

- Chapter 5 introduced the Tree-Transformer, a Transformer based approach which operates on tree structured data. We discussed the architecture of the system, including the addition of

the Tree Convolution Block which allows the network to operate on the tree structure. We also illustrated the necessity for encoder-decoder attention in allowing the decoder access to the encoder, and for self-attention which enforces conditional ordering of nodes in the tree. We then tested the system on the SATE IV dataset for code correction, illustrating significantly improved results over sequence based methods. We also tested the tree-transformer on benchmark datasets for grammar error correction, showing improved performance over passed models, especially in recall.

6.2 Strengths, Weaknesses, and Further Research

6.2.1 Improved Generation of Discrete Sequences using Wasserstein Generative Adversarial Networks

Using the method presented in chapter 3 we are able to produce realistic sentences using a GAN without pre-training. To the best of our knowledge, this is the first time this has been accomplished. However, one major downside to our approach is that it produces samples (e.g. one-hot outputs) instead of probabilities. As such we cannot compute the probability of a particular output, nor use those probabilities to compute metrics like perplexity, which are common for language modeling tasks. One potential solution for this is a transfer learning approach, which uses our trained generator to train a new network which does produce probabilities. This could be achieved by using the samples outputted from our generator as the desired output from a new neural language model. Training in this way can be done quickly by initializing the new neural language model's parameters to those of the generator, excluding the parameters used for added noise.

The ability to train without any pre-training is an important aspect to showing the robustness of our WGAN method. However, our results are likely to improve with pre-training. Additionally, our method may have difficulty competing with the standard neural language model approach, which can be trained to output probabilities directly from monolingual data. The model in this chapter provides a necessary first step toward the work in chapter 4, which allows for the training of a neuro machine translation system in the absence of paired data.

6.3 Learning to Repair Software Vulnerabilities with Generative Adversarial Networks

One of the major strong suits of the approach presented in chapter 4 is that we can use it on very large, unpaired datasets, which should be relatively easy to obtain. Additionally, since the model

matches the distributions of generated outputs to the distribution of real data it is fairly robust to false positives in the labeling of incorrect data. This allows us to consider groups of data as inputs. For example, a dataset could be formed by using sets of essays by native speakers as the desired data, and essays from non-native speakers as input data. Unfortunately, since data gathered in this way is unpaired it cannot be used to compare to current neural machine translation baselines. As such, in this thesis we chose datasets which have paired data for comparison, and leave unpaired datasets for future work.

For future work we would like to test our method on additional datasets and tasks. One such task is grammar error correction, of which their are several datasets available including CONLL 2014 and AESW, which were explored in chapter 5. Also, unpaired data gathering should also allow us to obtain datasets which are much larger than the grammar error correction datasets which are currently available.

Our approach allows us to train a NMT system in the absence of paired data, but does have some limitations, mainly because of its use of a GAN. Similarly to what was discussed in the previous section, the network does not produce probabilities, but instead samples. This makes it impossible to use techniques like beam search to produce the highest probability output, or to provide the top N corrections for a particular input. Additionally, because the system uses a WGAN it can be difficult to train, necessitating careful balancing of training iterations and learning rate between generator and discriminator. The results presented in this dissertation are the end result of many tuning runs, and achieving similar performance on new datasets will likely also necessitate a reasonably large amount of hyperparameter tuning.

6.4 Tree-Transformer: A Transformer-Based Method for Correction of Tree-Structured Data

One of the main advantages of the Tree-Transformer model presented in chapter 5 is its speed. The ability to compute all nodes in a layer in parallel results in speedup of several orders of magnitude over similar RNN or LSTM based approaches. Additionally, the transformer model is a highly used model in the literature, and is known to achieve very high accuracy, making the Tree-Transformer design more accessible and understandable for future researchers.

One downside of the Tree-Transformer approach is that it is dependent upon the ability to translate from sequence to tree and back again. In the task of grammar error correction this makes the networks performance dependent upon the parser used to translate sentences into parse trees.

Further more, since these parsers are generally only trained on correct sentences their behavior on incorrect sentences is undefined. This is mitigated somewhat by the plasticity of the network, which allows the network to learn some of the quirks and inaccuracies of the parser on incorrect data.

For the task of code correction the Tree-Transformer is dependent on the abstract syntax tree parser. However, parsing for code is less of a problem than for language because abstract syntax trees use a well defined grammar. Instead, the main difficulty of code correction is returning corrections made to the abstract syntax tree to source code. Some recent work has enabled source code to be updated from the abstract syntax tree [66, 67], but this is still an open research area and additional work is likely necessary before the Tree-Transformer could be used in production on code.

For future work, we would like to explore the use of the Tree-Transformer on language translation. Specifically, many language pairs still have very low-resource datasets, for example English to Romanian. These low-resource pairs could likely benefit from the Tree-Transformers ability to use the underlying grammar.

Bibliography

[1] J. Ji, Q. Wang, K. Toutanova, Y. Gong, S. Truong, and J. Gao. A Nested Attention Neural Hybrid Model for Grammatical Error Correction. *Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 753–762, 2017.

[2] Z. Yuan and T. Briscoe. Grammatical error correction using neural machine translation. *North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL HLT)*, 2016.

[3] A. Schmaltz, Y. Kim, A. Rush, and S. Shieber. Adapting Sequence Models for Sentence Correction. *Empirical Methods in Natural Language Processing (EMNLP)*, 2017.

[4] Z. Xie, A. Avati, N. Arivazhagan, D. Jurafsky, and A. Ng. Neural Language Correction with Character-Based Attention. *arXiv:1603.09727*, March 2016.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need. *Neural Information Processing Systems (NIPS)*, 2017.

[6] D. Bahdanau, K. Cho, and Y. Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *International Conference on Learning Representations (ICLR)*, 2015.

[7] M.-T. Luong, H. Pham, and C. Manning. Effective Approaches to Attention-based Neural Machine Translation. *Empirical Methods in Natural Language Processing (EMNLP)*, 2015.

[8] A short history of translation through the ages. `https://marielebert.wordpress.com/2016/11/02/translation/`.

[9] A brief history of Natural Language Processing. `http://www.cs.bham.ac.uk/~pjh/sem1a5/pt1/pt1_history.html`.

[10] G. Hendrix, E. Sacerdoti, D. Sagalowicz, and J. Slocum. Developing a Natural Language Interface to Complex Data. *ACM Trans. Database Syst.*, 1978.

[11] R. Brachman and J. Schmolze. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9:171–216, 1985.

[12] M. Gales and S. Young. The Application of Hidden Markov Models in Speech Recognition. *Foundations and Trends in Signal Processing*, 2007.

[13] M. Junczys-Dowmunt, R. Grundkiewicz, S. Guha, and K. Heafield. Approach Neural Grammatical Error Correction as a Low-Resource Machine Translation Task. *North American Chapter of the Association of Computational Linguistics (NAACL-HLT)*, 2018.

[14] S. Hochreiter. The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 06(02):107–116, April 1998.

[15] S. Hochreiter and J. Schmidhuber. Long Short-term Memory. *Neural Computation*, 9(8):1735–1780, December 1997.

[16] R. Williams and D. Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, 1989.

[17] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *Neural Information Processing Systems (NIPS)*, June 2014.

[18] M. Arjovsky and L. Bottou. Towards Principled Methods for Training Generative Adversarial Networks. *International Conference on Learning Representations (ICLR)*, 2017.

[19] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein Generative Adversarial Networks. *International Conference on Machine Learning (ICML)*, 2017.

[20] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville. Improved Training of Wasserstein GANs. *Neural Information Processing Systems (NIPS)*, 2017.

[21] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *International Conference on Learning Representations (ICLR)*, 2016.

[22] T. Mikolov. *Statistical Language Models based on Neural Networks*. PhD thesis, Brno University of Technology, 2012.

[23] L. Yu, W. Zhang, J. Wang, and Y. Yu. SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient. *Association for the Advancement of Artifical Intelligence (AAAI)*, 2017.

[24] S. Rajeswar, S. Subramanian, F. Dutil, C. Pal, and A. Courville. Adversarial Generation of Natural Language. *2nd Workshop on Representation Learning for NLP (RepL4NLP)*, 2017.

[25] O. Press, A. Bar, B. Bogin, J. Berant, and L. Wolf. Language Generation with Recurrent Generative Adversarial Networks without Pre-training. *1st Workshop on Subword and Character Level Models in NLP (SCLeM)*, 2017.

[26] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson. One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling. *arXiv.org*, 2013.

[27] G. Lample, L. Denoyer, and M. Ranzato. Unsupervised Machine Translation Using Monolingual Corpora Only. *International Conference on Learning Representations (ICLR)*, 2018.

[28] A. Gomez, S. Huang, I. Zhang, B. Li, M. Osama, and L. Kaiser. Unsupervised Cipher Cracking Using Discrete GANs. *International Conference on Learning Representations (ICLR)*, 2018.

[29] J. Harer, O. Ozdemir, T. Lazovich, C. P. Reale, R. L. Russell, L. Y. Kim, and P. Chin. Learning to Repair Software Vulnerabilities with Generative Adversarial Networks. *Neural Information Processing Systems (NeuroIPS)*, 2018.

[30] J. Li, W. Monroe, T. Shi, S. Jean, A. Ritter, and D. Jurafsky. Adversarial Learning for Neural Dialogue Generation. *Empirical Methods in Natural Language Processing (EMNLP)*, 2017.

[31] Hjelm, R Devon, Jacob, Athul Paul, Che, Tong, Trischler, Adam, Cho, Kyunghyun, and Bengio, Yoshua. Boundary-Seeking Generative Adversarial Networks. *International Conference on Learning Representations (ICLR)*, 2018.

[32] E. Jang, S. Gu, and B. Poole. Categorical Reparameterization with Gumbel-Softmax. *International Conference on Learning Representations (ICLR)*, 2017.

[33] C. Maddison, A. Mnih, and Y. Teh. The Concrete Distribution - A Continuous Relaxation of Discrete Random Variables. *International Conference on Learning Representations (ICLR)*, 2017.

[34] Y. Zhang, Z. Gan, K. Fan, Z. Chen, R. Henao, D. Shen, and L. Carin. Adversarial Feature Matching for Text Generation. *International Conference on Machine Learning (ICML)*, 2017.

[35] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space. *arXiv.org*, 2013.

[36] R. Sennrich, B. Haddow, and A. Birch. Neural Machine Translation of Rare Words with Subword Units. *Association of Computational Linguistics (ACL)*, 2016.

[37] M. Monperrus. Automatic Software Repair: A Bibliography. *ACM Computing Surveys (CSUR)*, 2018.

[38] X. B. Le, D. Lo, and C. Le Goues. History driven program repair. *Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[39] F. Long and M. Rinard. Automatic patch generation by learning correct code. *Principles of Programming Languages (POPL)*, 2016.

[40] Devlin, Jacob, Uesato, Jonathan, Singh, Rishabh, and Kohli, Pushmeet. Semantic Code Repair using Neuro-Symbolic Transformation Networks. *arXiv:1710.11054*, October 2017.

[41] R. Gupta, S. Pal, A. Kanade, and S. Shevade. DeepFix: Fixing Common C Language Errors by Deep Learning. *Association for the Advancement of Artifical Intelligence (AAAI)*, pages 1345–1351, 2017.

[42] M. Mirza and S. Osindero. Conditional Generative Adversarial Nets. *arXiv:1411.1784*, November 2014.

[43] Z. Yang, W. Chen, F. Wang, and B. Xu. Improving Neural Machine Translation with Conditional Sequence Generative Adversarial Nets. *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2018.

[44] J.-Y. Zhu, T. Park, P. Isola, and A. Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. *International Conference on Computer Vision (ICCV)*, 2017.

[45] A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb. Learning from Simulated and Unsupervised Images through Adversarial Training. *Computer Vision and Pattern Recognition (CVPR)*, 2017.

[46] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. *International Conference on Machine Learning (ICML)*, 2008.

[47] V. Okun, A. Delaitre, and P. Black. Report on the Static Analysis Tool Exposition (SATE) IV. Technical report, 2013.

[48] S. Chollampatt and H. Ng. A Multilayer Convolutional Encoder-Decoder Neural Network for Grammatical Error Correction. *Association for the Advancement of Artificial Intelligence (AAAI)*, 2018.

[49] K. S. Tai, R. Socher, and C. Manning. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *53rd Annual Meeting of the Association for Computational Linguistics and The 7th International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, 2015.

[50] X. Zhu, P. Sobhani, and H. Guo. Long Short-Term Memory Over Tree Structures. *International Conference on Machine Learning (ICML)*, 2015.

[51] R. Socher, C. Lin, A. Ng, and C. Manning. Parsing Natural Scenes and Natural Language with Recursive Neural Networks. *International Conference on Machine Learning (ICML)*, 2011.

[52] A. Eriguchi, K. Hashimoto, and Y. Tsuruoka. Tree-to-Sequence Attentional Neural Machine Translation. *Association of Computational Linguistics (ACL)*, 2016.

[53] L. Dong and M. Lapata. Language to Logical Form with Neural Attention. *Association of Computational Linguistics (ACL)*, 2016.

[54] D. Alvarez-Melis and T. Jaakkola. Tree-Structured Decoding with Doubly-Recurrent Neural Networks. *International Conference on Learning Representations (ICLR)*, 2017.

[55] O. Vinyals, L. Kaiser, T. Koo, S. Petrov, I. Sutskever, and G. Hinton. Grammar as a Foreign Language. *Neural Information Processing Systems (NIPS)*, 2015.

[56] R. Aharoni and Y. Goldberg. Towards String-to-Tree Neural Machine Translation. *Association of Computational Linguistics (ACL)*, 2017.

[57] M. Rabinovich, M. Stern, and D. Klein. Abstract Syntax Networks for Code Generation and Semantic Parsing. *Association of Computational Linguistics (ACL)*, 2017.

[58] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-Symbolic Program Synthesis. *International Conference on Learning Representations (ICLR)*, 2017.

[59] P. Yin and G. Neubig. A Syntactic Neural Model for General-Purpose Code Generation. *Association of Computational Linguistics (ACL)*, 2017.

[60] X. Zhang, L. Lu, and M. Lapata. Top-down Tree Long Short-Term Memory Networks. *North American Chapter of the Association of Computational Linguistics (NAACL)*, 2016.

[61] X. Chen, C. Liu, and D. Song. Tree-to-tree Neural Networks for Program Translation. *Neural Information Processing Systems (NeurIPS)*, 2018.

[62] S. Chakraborty, M. Allamanis, and B. Ray. Tree2Tree Neural Translation Model for Learning Source Code Changes. *arXiv pre-print*, 2018.

[63] M. Junczys-Dowmunt and R. Grundkiewicz. Phrase-based Machine Translation is State-of-the-Art for Automatic Grammatical Error Correction. *Empirical Methods in Natural Language (EMNLP)*, 2016.

[64] S. Chollampatt and H. T. Ng. Connecting the Dots: Towards Human-Level Grammatical Error Correction. *The 12th Workshop on Innovative Use of NLP for Building Educational Applications. Association for Computational Linguistics (ACL)*, 2017.

[65] D. Dahlmeier, H. Ng, and S. Wu. Building a Large Annotated Corpus of Learner English - The NUS Corpus of Learner English. *North American Chapter of the Association of Computational Linguistics (NAACL)*, 2013.

[66] C. Lattner and V. Adve. LLVM - A Compilation Framework for Lifelong Program Analysis & Transformation. *CGO*, 2004.

[67] Clang Library. `clang.llvm.org`.

[68] Offical scorer for CoNLL 2014 Shared Task. `https://github.com/nusnlp/m2scorer/releases`, 2014.

[69] C. Goller and A. Kuchler. Learning task-dependent distributed representations by backpropagation through structure. *International Conference on Neural Networks (ICNN'96)*, 1996.

[70] D. Chen and C. Manning. A Fast and Accurate Dependency Parser using Neural Networks. *Emperical Methods in Natural Language Processing (EMNLP)*, 2014.

[71] R. Socher, J. Bauer, C. Manning, and A. Ng. Parsing With Compositional Vector Grammars. *Association for Computational Linguistics (ACL)*, 2013.

[72] D. Klein and C. Manning. Accurate Unlexicalized Parsing. *Association for Computational Linguistics (ACL)*, 2003.

[73] Shift-Reduce Constituency Parser. `https://nlp.stanford.edu/software/srparser.html`.

[74] Penn Treebank II Tags. `https://web.archive.org/web/20130517134339/http://bulba.sdsu.edu/jeanette/thesis/PennTags.html`.

[75] B. Heinzerling and M. Strube. BPEmb: Tokenization-free Pre-trained Subword Embeddings in 275 Languages. In N. C. C. chair), K. Choukri, C. Cieri, T. Declerck, S. Goggi, K. Hasida, H. Isahara, B. Maegaard, J. Mariani, H. Mazo, A. Moreno, J. Odijk, S. Piperidis, and T. Tokunaga, editors, *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, May 7-12, 2018 2018. European Language Resources Association (ELRA).

[76] H. Ng, S. Wu, T. Briscoe, C. Hadiwinoto, R. Susanto, and C. Bryant. The CoNLL-2014 Shared Task on Grammatical Error Correction. *Conference on Computational Natural Language Learning, Association for Computational Linguistics (ACL)*, 2014.

[77] V. Daudaravicius, R. Banchs, E. Volodine, and C. Napoles. A report on the automatic evaluation of scientific writing shared task. *11th Workshop on Innovative Use of NLP for Building Educational Applications, Association for Computational Linguistics (ACL)*, 2016.

[78] Apted Python Library. `https://pypi.org/project/apted/`.

[79] M. Pawlik and N. Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems 56*, 2016.

[80] M. Pawlik and N. Augsten. Efficient Computation of the Tree Edit Distance. *ACM Transactions on Database Systems*, 30:2, 2015.

CURRICULUM VITAE