Learning to Generate Textual Data

Guillaume Bouchard[†‡*] and **Pontus Stenetorp**[†*] and **Sebastian Riedel**[†]
{g.bouchard,p.stenetorp,s.riedel}@cs.ucl.ac.uk
[†]Department of Computer Science, University College London
[‡]Bloomsbury AI

Abstract

To learn text understanding models with millions of parameters one needs massive amounts of data. In this work, we argue that generating data can compensate for this need. While defining generic data generators is difficult, we propose to allow generators to be "weakly" specified in the sense that a set of parameters controls how the data is generated. Consider for example generators where the example templates, grammar, and/or vocabulary is determined by this set of parameters. Instead of manually tuning these parameters, we learn them from the limited training data at our disposal. To achieve this, we derive an efficient algorithm called GeneRe that jointly estimates the parameters of the model and the undetermined generation parameters. We illustrate its benefits by learning to solve math exam questions using a highly parametrized sequence-to-sequence neural network.

1 Introduction

Many tasks require a large amount of training data to be solved efficiently, but acquiring such amounts is costly, both in terms of time and money. In several situations, a human trainer can provide their domain knowledge in the form of a generator of virtual data, such as a negative data sampler for implicit feedback in recommendation systems, physical 3D rendering engines as a simulator of data in a computer vision system, simulators of physical processes to solve science exam question, and math problem generators for automatically solving math word problems.

[*] Contributed equally to this work.

Domain-specific data simulators can generate an arbitrary amount of data that can be treated exactly the same way as standard observations, but since they are virtual, they can also be seen as regularizers dedicated to the task we want to solve (Scholkopf and Smola, 2001). While simple, the idea of data simulation is powerful and can lead to significantly better estimations of a predictive model because it prevents overfitting. At the same time it is subject to a strong model bias, because such data generators often generate data that is different from the observed data.

Creating virtual samples is strongly linked to transfer learning when the task to transfer is correlated to the objective (Pan and Yang, 2010). The computer vision literature adopted this idea very early through the notion of *virtual samples*. Such samples have a natural interpretation: by creating artificial perturbations of an image, its semantics is likely to be unchanged, i.e. training samples can be rotated, blurred, or slightly cropped without changing the category of the objects contained in the image (Niyogi et al., 1998).

However, for natural language applications the idea of creating invariant transformations is difficult to apply directly, as simple meaning-preserving transformations – such as the replacement of words by their synonyms or active-passive verb transformations – are quite limited. More advanced meaning-preserving transformations would require an already good model that understands natural language. A more structure-driven approach is to build top-down generators, such as probabilistic grammars, with a much wider coverage of linguistic phenomena. This way of being able to leverage many years of research in computational linguistics to create good data generators would be a natural and useful reuse of scientific knowledge, and better than blindly believing in the current trend of "data takes all".

While the idea of generating data is straightforward, one could argue that it may be difficult to come up with good generators. What we mean by a good generator is the ability to help predicting test data when the model is trained on the generated data. In this paper, we will show several types of generators, some contributing more than others in their ability to generalize to unseen data. When designing a good generator there are several decisions one must make: should we generate data by modifying existing training samples, or "go wild" and derive a full probabilistic context-free grammar that could possibly generate unnatural examples and add noise to the estimator? While we do not arrive at a specific framework to build programs that generate virtual data, in this work we assume that a domain expert can easily write a program in a programming language of her choice, leaving some generation parameters unspecified. In our approach these unspecified parameters are automatically learned, by selecting the ones most compatible with the model and the training data.

In the next section, we introduce GENERE, a generic algorithm that extends any gradient-based learning approach with a data generator that can be tuned while learning the model on the training data using stochastic optimization. In Section 2.2, we show how GENERE can be adapted to handle a (possibly non-differentiable) black-box sampler without requiring modifications to it. We also illustrate how this framework can be implemented in practice for a specific use case: the automatic solving of math exam problems. Further discussion is given in the concluding section.

2 Regularization Based on a Generative Model

As with any machine learning approach, we assume that given the realisation of a variable $x \in \mathcal{X}$ representing the input, we want to predict the distribution of a variable $y \in \mathcal{Y}$ representing the output. The goal is to find this predictive distribution by learning it from examples $\mathcal{D} := \{(x_i, y_i)\}_{i=1}^n$.

Building on the current success in the application of deep learning to NLP, we assume that there exists a good model family $\{f_\theta, \theta \in \Theta\}$ to predict y given x, where θ is an element of the parameter space Θ. For example, the stacked LSTM encoder-decoder is a general purpose model that has helped to improve results on relatively complex tasks, such as machine translation (Sutskever et al., 2014), syntactic parsing (Vinyals et al., 2014), semantic parsing (Dong and Lapata, 2016) and textual entailment (Rocktäschel et al., 2016).

For many applications, the amount of training data is too small or too costly to acquire. We hence look for alternative ways to regularize the model so that we can achieve good performance using few data points.

Let $p_\theta(y|x)$ be the target prediction model. Given the training dataset $\mathcal{D}$, the penalized maximum likelihood estimator is obtained by $\min_{\theta\in\Theta} \mathcal{L}(\theta)$ where:

$$\mathcal{L}(\theta) := \ell(\theta) + \lambda\Omega(\theta) \ . \qquad (1)$$

where $\ell(\theta) := -\frac{1}{n}\sum_{i=1}^n \log p_\theta(y_i|x_i) = \mathbb{E}_{\hat{P}}\left[\log p_\theta(y|x)\right]$ is the negative log-likelihood. Here, $\Omega(\theta)$ is a regularizer that prevents over-fitting, $\lambda \in \mathbb{R}$ the regularization parameter that can be set by cross-validation, and $\hat{P}$ is the empirical distribution. Instead of using a standard regularizer Ω – such as the squared norm or the Lasso penalty which are domain-agnostic, – in this paper we propose to use a generative model to regularize the estimator.

Domain knowledge A natural way to inject background knowledge is to define a generative model that simulates the way the data is generated. In text understanding applications, such generative models are common and include probabilistic context-free grammars (PCFG) and natural language generation frameworks (e.g. SimpleNLG (Gatt and Reiter, 2009)). Let $P_\gamma(x, y)$ be such a generative model parametrized by a continuous parameter vector $\gamma \in \Gamma$, such as the concatenation of all the parameters of the production rules in a PCFG. One important difference between the discriminative and the generative probability distributions is that the inference problem of y given x might be intractable[1] for the generative model, even if the joint model can be computed efficiently.

In this work, we use the following regularizer:

$$\Omega(\theta) \quad := \quad \min_{\gamma\in\Gamma} \mathbb{E}_{P_\gamma(x,y)} \left[\log \left(\frac{P_\gamma(y|x)}{p_\theta(y|x)} \right) \right] \; .(2)$$

This regularizer makes intuitive sense as it corresponds to the smallest possible Kullback-Leibler divergence between the generative and discriminative models. We can see that if the generator p_γ is close to the distribution that generates the test data, the method can potentially yield good performance. However, in practice, γ is unknown and difficult to set. In this work, we focus on several techniques that can be used to estimate the generative parameter vector γ on the training data, making the regularizer data-dependent.

Minimizing the objective from Equation (1) is equivalent to minimize the following function over $\Theta \times \Gamma$:

$$\mathcal{L}(\theta,\gamma) \quad := \quad \ell(\theta) + \lambda \mathbb{E}_{P_\gamma(x,y)} \left[\log \left(\frac{p_\gamma(y|x)}{p_\theta(y|x)} \right) \right] \quad .$$

This estimator is called GeneRe for *Generative Regularization* and can be viewed as a Generative-Discriminative Tradeoff estimator (GDT (Bouchard and Triggs, 2004)) that smoothly interpolates between a purely un-regularized discriminative model when $\lambda = 0$ and a generative model when λ tends to infinity.

2.1 The GeneRe Algorithm

The objective $\mathcal{L}(\theta,\gamma)$ can also be written as an expectation under a mixture distribution $\tilde{P}_\gamma := \frac{1}{1+\lambda}\hat{P} + \frac{\lambda}{1+\lambda}P_\gamma$. The two components of this mixture are the empirical data distribution $\hat{P}$ and the generation distribution P_γ. The final objective is penalized by the entropy of the the generation $\mathcal{H}(\gamma) := \mathbb{E}_{P_\gamma}\left[\log p_\gamma(y|x)\right]$:

$$\mathcal{L}(\theta,\gamma) = -(1+\lambda)\mathbb{E}_{\tilde{P}_\gamma}\left[\log p_\theta(y|x)\right] - \lambda\mathcal{H}(\gamma) \quad . \quad (3)$$

This objective can be minimized using stochastic gradient descent by sampling real data or generated data according to the proportions $\frac{1}{1+\lambda}$ and $\frac{\lambda}{1+\lambda}$, respectively. The pseudocode is provided in Algorithm 1. It can be viewed as a variant of the Reinforce algorithm which is commonly used in Reinforcement Learning (Williams, 1988) using the policy gradient. It is straightforward to verify that at each iteration, GeneRe computes a noisy estimate of the exact gradient of the objective function $\mathcal{L}(\theta,\gamma)$ with respect to the model parameters θ and the generation parameters[2] γ.

An important quantity introduced in Algorithm 1 is the baseline value μ that approximates the average log-likelihood of a point sampled according to $\tilde{P}_\gamma$. Since it is unknown in general, an average estimate is obtained using a geometric averaging scheme with a coefficient α that is typically set to 0.98.

Algorithm 1 The GeneRe Algorithm

```
Require: $\hat{P}$: real data sampler
Require: $P_\gamma$: parametric data generator
Require: $\lambda$: generative regularization strength
Require: $\eta$: learning rate
Require: $\alpha$: baseline smoothing coefficient
 1: Initialize parameters $\theta$, sampling coefficients $\gamma$
    and baseline $\mu$
 2: for $t = 1, 2, \cdots$ do
 3:     $x, y \sim \frac{1}{1+\lambda}\hat{P} + \frac{\lambda}{1+\lambda}P_\gamma$
 4:     $g_\theta \leftarrow \nabla_\theta \log p_\theta(y|x)$
 5:     $g_\gamma \leftarrow (\log p_\theta(y|x) - \mu)\, \nabla_\gamma \log p_\gamma(x,y)$
 6:     $(\theta,\gamma) \leftarrow (\theta,\gamma) - \eta(g_\theta, g_\gamma)$
 7:     $\mu \leftarrow \alpha\mu + (1-\alpha)\log p_\theta(y|x)$
 8: end for
```

Generative models: interpretable sampling, intractable inference Generative modeling is natural because we can consider latent variables that add interpretable meaning to the different components of the model. For example, in NLP we can define the latent variable as being the relations that are mentioned in the sentence.

[1]Even if tractable, inference can be very costly: for example, PCFG decoding can be done using dynamic programming and has a cubic complexity in the length of the decoded sentence, which is still too high for some applications with long sentences.

[2]The derivative with respect to γ, leads to Algorithm 1 with $\mu = -1$, but the algorithm is also valid for different values of μ as the average gradient remains the same if we add a multiple of $\nabla_\gamma \log p_\gamma(x,y)$ to the gradient g_γ (line 5 in Algorithm 1) which has zero-mean on average. Choosing μ to be the average of the past gradient enables the gradient to have a lower variance.

We could consider two main types of approaches to choose a good structure for a parameterized data generator:

- Discrete data structure: we can use efficient algorithms, such as dynamic programming to perform sampling and which can propagate the gradient
- Continuous distribution: having a continuous latent variable enables easy handling of correlations across different parts of the model.

It is often laborious to design data generators which can return the probability of the samples it generates[3], as well as the gradient of this probability with respect to the input parameters γ.

In the next section, we show how to alleviate this constraint by allowing any data-generating code to be used with nearly no modification.

2.2 GENERE with a Black Box Sampler

Let us assume the data generator is a black box that takes a K-dimensional seed vector as input and outputs an input-output sample x, y. To enable GENERE to be applied without having to modify the code of data generators. The trick is to use a existing generator with parameter γ, and to create a new generator that essentially adds noise to γ. This noise will be denoted $\Delta \in \Gamma$. We used the following data generation process:

1. Sample a Gaussian seed vector $\Delta \sim \mathcal{N}(0, I)$
2. Use the data generator G_z with seed value $z := \Delta + \gamma$ to generate an input-output sample (x, y).

This two-step generation procedure enables the gradient information to be computed using the density of a Gaussian distribution. The use of a standardized centered variable for the seed is justified by the fact that the parametrization of G_z takes into account possible shifts and rescaling. Formally, this is equivalent to Algorithm 1 with the following generative model:

$$p_\gamma(x, y) = \mathbb{E}_{\Delta \sim \mathcal{N}(0,I)} \left[g_{\gamma+\Delta}(x, y) \right] \tag{4}$$

[3]This difficulty comes from the fact that generators may be using third-party code, such as rendering engines, grammars sampler, and deterministic operations such a sorting that are non-differentiable.

where g_z is the density of the black-box data generator G_z for the seed value $z \in \mathbb{R}^K$. Ideally, the second data generator that takes z as an input and generates the input/output pair (x, y) should be close to a deterministic function in order to allocate more uncertainty in the trainable part of the model which corresponds to the Gaussian distribution.[4]

Learning The pseudo-code for the Black Box GENERE variant is shown in Algorithm 2. It is similar to Algorithm 1, but here the sampling phase is decomposed into the two steps: A random Gaussian variable sampling followed by the black box sampling of generators.

Algorithm 2 Black Box GENERE

```
Require: $\hat{P}$: real data sampler
Require: $G(\gamma)$: black box data generator
Require: $\lambda$: generative regularization strength
Require: $\eta_\gamma, \eta_\theta$: learning rates
 1: Initialize parameters $\theta$, sampling coefficients $\gamma$
    and baseline $\mu$
 2: for $t = 1, 2, \cdots$ do
 3:     if $\frac{1}{1+\lambda} > \mathcal{U}([0, 1])$ then
 4:         $x, y \sim \hat{P}$
 5:     else
 6:         $\Delta \sim \mathcal{N}(\mathbf{0}, I)$
 7:         $x, y \sim G_{\gamma+\Delta}$
 8:         $\gamma \leftarrow \gamma - \eta_\gamma(\log p_\theta(y|x) - \mu)\Delta$
 9:     end if
10:     $\theta \leftarrow \theta - \eta_\theta \nabla_\theta \log p_\theta(y|x)$
11:     $\mu \leftarrow \alpha\mu + (1 - \alpha) \log p_\theta(y|x)$
12: end for
```

3 Application to Encoder-Decoder

In this section, we show that the GENERE algorithm is well suited to tune data generators for problems that are compatible with the encoder-decoder architecture commonly used in NLP.

3.1 Mixture-based Generators

In the experiments below, we consider mixture-based generators with known components but unknown mixture proportions. Formally, we

[4]What we mean by deterministic is that the black-box sampler has the form $\delta\{f(\Delta + \gamma) = (x, y)\}$, where δ is the indicator function.

parametrize the proportions using a softmax link $\sigma(t) := \exp(t_k)/\sum_{k'=1}^{K}\exp(t_{k'})$. In other words, the data generator distribution is:

$$p_\gamma(x,y) = \sum_{k=1}^{K} \sigma_k(\gamma+\Delta) p_k(x,y),$$

where $p_k(x,y)$ are data distributions, called *base generators*, that are provided by domain experts, and Δ is a K-dimensional centered Gaussian with an identity covariance matrix. This class of generator makes sense in practice, as we typically build multiple base generators $p_k(x,y)$, $k = 1,\cdots,K$, without knowing ahead of time which one is the most relevant. Then, the training data is used by the GENERE algorithm to automatically learn the optimal parameter γ that controls the contribution $\{\pi_k\}_{k=1}^{K}$ of each of the base generators, equal to $\pi_k := \mathbb{E}_{\Delta\sim\mathcal{N}(\mathbf{0},I)}\left[\sigma_k(\gamma+\Delta)\right]$.

3.2 Synthetic Experiment

In this section, we illustrate how GENERE can learn to identify the correct generator, when the data generating family is a mixture of multiple data generators and only one of these distributions – say p_1 – has been used to generate the data. The other distributions $(p_2,\cdots,p_K)$ are generating input-output data samples (x,y) with different distributions.

We verified that the algorithm correctly identifies the correct data distribution, and hence leads to better generalization performances than what the model achieves without the generator.

In this illustrative experiment, a simple text-to-equation translation problem is created, where inputs are sentences describing an equation such as "compute one plus three minus two", and outputs are symbolic equations, such as "X = 1 + 3 - 2". Numbers were varying between -20 and 20, and equations could have 2 or 3 numbers with 2 or 3 operations.

As our model, we used a 20-dimensional sequence-to-sequence model with LSTM recurrent units. The model was initialized using 200 iterations of standard gradient descent on the log-probability of the output. GENERE was run for 500 iterations, varying the fraction of real and generated samples from 0% to 100%. A ℓ_2 regularization of magnitude 0.1 was applied to the model. The baseline smoothing coefficient was set to 0.98 and the shrinkage parameter was set to 0.99. All the experiments were repeated 10 times and a constant learning rate of 0.1 was used.

Results are shown on Figure 1, where the average loss computed on the test data is plotted against the fraction of real data used during learning.

We can see that the best generalization performance is obtained when there is a balanced mix of real and artificial data, but the proportion depends on the amount of training data: on the left hand side, the best performance is obtained with generated data only, meaning that the number of training samples is so small that GENERE only used the training data to select the best base generator (the component p_1), and the best performance is attained using only generated data. The plot on the right hand side is interesting because it contains more training data, and the best performance is not obtained using only the generator, but with 40% of the real data, illustrating the fact that it is beneficial to jointly use real and simulated data during training.

3.3 Math word problems

To illustrate the benefit of using generative regularization, we considered a class of real world problems for which obtaining data is costly: learning to answer math exam problems. Prior work on this problem focuses on standard math problems given to students aged between 8 and 10, such as the following:[5]

> For Halloween Sarah received 66 pieces of candy from neighbors and 15 pieces from her older sister. If she only ate 9 pieces a day, how long would the candy last her?

The answer is given by the following equation: $\mathtt{X} = (66+15)/9$. Note that similarly to real world school exams, giving the final answer of (9 in this case) is not considered enough for the response to be correct.

The only publicly available word problem datasets we are aware of contain between 400 and 600 problems (see Table 2), which is not enough to properly train sufficiently rich models that capture the link between the words and the quantities involved in the problem.

[5]From the Common Core dataset (Roy and Roth, 2015)

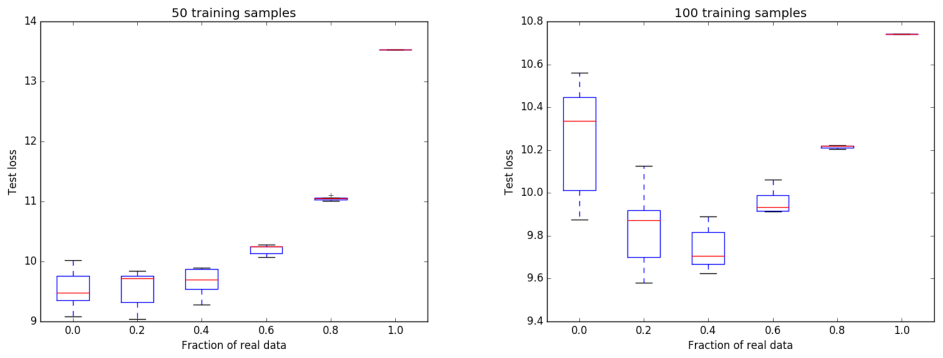

Figure 1: Test loss vs. fraction of real data used in GENERE on the text-to-equation experiment.

Sequence-to-sequence learning is the task of predicting an output sequence of symbols based on a sequence of input symbols. It is tempting to cast the problem of answering math exams as a sequence-to-sequence problem: given the sequence of words from the problem description, we can predict the sequence of symbols for the equation as output. Currently, the most successful models for sequence prediction are Recurrent Neural Nets (RNN) with non-linear transitions between states.

Treated as a translation problem, math word problem solving should be simpler than developing a machine translation model between two human languages, as the output vocabulary (the math symbols) is significantly smaller than any human vocabulary. However, machine translation can be learned on millions of pairs of already translated sentences, and such massive training datasets dwarf all previously introduced math exam datasets.

We used standard benchmark data from the literature. The first one, AI2, was introduced by Hosseini et al. (2014) and covers addition and subtraction of one or two variables or two additions scraped from two web pages. The second (IL), introduced by Roy et al. (2015), contains single operator questions but covers addition, subtraction, multiplication, and division, and was also obtained from two, although different from AI2, web pages. The last data set (CC) was introduced by Roy and Roth (2015) to cover combinations of different operators and was obtained from a fifth web page.

An overview of the equation patterns in the data is shown in Table 1. It should be noted that there are sometimes numbers mentioned in the problem description that are not used in the equation.

AI2	IL	CC
$\mathtt{X}+\mathtt{Y}$	$\mathtt{X}+\mathtt{Y}$	$\mathtt{X}+\mathtt{Y}-\mathtt{Z}$
$\mathtt{X}+\mathtt{Y}+\mathtt{Z}$	$\mathtt{X}-\mathtt{Y}$	$\mathtt{X}*(\mathtt{Y}+\mathtt{Z})$
$\mathtt{X}-\mathtt{Y}$	$\mathtt{X}*\mathtt{Y}$	$\mathtt{X}*(\mathtt{Y}-\mathtt{Z})$
	$\mathtt{X}/\mathtt{Y}$	$(\mathtt{X}+\mathtt{Y})/\mathtt{Z}$
		$(\mathtt{X}-\mathtt{Y})/\mathtt{Z}$

Table 1: Patterns of the equations seen in the datasets for one permutation of the placeholders.

	AI2	IL	CC
Train	198	214	300
Dev	66	108	100
Test	131	240	200
Total	395	562	600

Table 2: Math word problems dataset sizes.

As there are no available train/dev/test splits in the literature we introduced such splits for all three data sets. For AI2 and CC, we simply split the data randomly and for IL we opted to maintain the clusters described in Roy and Roth (2015). We then used the implementation of Roy and Roth (2015) provided by the authors, which is the current state-of-the-art for all three data sets, to obtain results to compare our model against. The resulting data sizes are shown on Table 2. We verified that there are no duplicate problems, and our splits and a fork of the baseline implementation are available online.[6]

3.4 Development of the Generator

Generators were organized as a set of 8 base generators p_k, summarized in Table 4. Each base generator

[6]`https://github.com/ninjin/roy_and_roth_2015`

John sprints to William's apartment. The distance is 32 yards from John's apartment to William's apartment. It takes John 2 hours to at the end get there. How fast did John go?	32 / 2
Sandra has 7 erasers. She grasps 7 more. The following day she grasps 18 whistles at the local supermarket. How many erasers does Sandra have in all?	7 + 7
A pet store had 81 puppies In one day they sold 41 of them and put the rest into cages with 8 in each cage. How many cages did they use?	(81 - 41) / 8
S1 V1 Q1 O1 C1 S1(pronoun) V2 Q2 of O1(pronoun) and V2 the rest into O3(plural) with Q3 in each O3. How many O3(plural) V3?	(Q1 - Q2) / Q3

Table 3: Examples of generated sentences (first 3 rows). The last row is the template used to generate the 3rd example where brackets indicate modifiers, symbols starting with 'S' or 'O' indicate a noun phrase for a subject or object, symbols with 'V' indicate a verb phrase, and symbols with 'Q' indicate a quantity. They are identified with a number to match multiple instances of the same token.

has several functions associated with it. The functions were written by a human over 3 days of full-time development. The first group of base generators is only based on the type of symbol the equation has, the second group is the pair (#1, #2) to represent equations with one or two symbols. Finally, the last two generators are more experimental as they correspond to simple modifications applied to the available training data. The Noise 'N' generator picks one or two random words from a training sample to create a new (but very similar) problem. Finally, the 'P' generator is based on computing the statistics of the words for the same question pattern (as one can see in Table 1), and generates data using simple biased word samples, where words are distributed according to their average positions in the training data (positions are computed relatively to the quantities appearing in the text, i.e. "before the first number", "between the 1st and the 2nd number", etc.).

3.5 Implementation Details

We use a standard stacked RNN encoder-decoder (Sutskever et al., 2014), where we varied the recurrent unit between LSTM and GRU (Cho et al., 2014), stack depth from 1 to 3, the size of the hidden states from 4 to 512, and the vocabulary threshold size. As input to the encoder, we downloaded pre-trained 300-dimensional embeddings trained on Google News data using the word2vec software (Mikolov et al., 2013). The development data was used to tune these parameters before performing the evaluation on the test set. We obtained the best performances with a single stack, GRU units, and a hidden state size of 256.

	The problem...
+	contains at least one addition
-	contains at least one subtraction
*	contains at least one multiplication
/	contains at least one division
1	has a single mathematical operation
2	has a couple of mathematical operations
N	is a training sample with words removed
P	is based on word position frequencies

Table 4: The base generators to create math exam problems.

The optimization algorithm was based on stochastic gradient descent using Adam as an adaptive step size scheme (Kingma and Ba, 2014), with mini-batches of size 32. A total of 256 epochs over the data was used in all the experiments.

To evaluate the benefit of learning the data generator, we used a hybrid method as a baseline where a fraction of the data is real and another fraction is generated using the default parameters of the generators (i.e. a uniform distribution over all the base generators). The optimal value for this fraction obtained on the development set was 15% real data, 85% generated data. For GENERE, we used a fixed

	AI2	IL	CC	Avg.
RR2015	82.4	75.4	55.5	71.1
100% Data	72.5	53.7	95.0	73.7
100% Gen	60.3	51.2	92.0	67.8
85%Gen + 15%Data	74.0	55.4	97.5	75.6
GENERE	77.9	56.7	98.5	77.7

Table 5: Test accuracies of the math-exam methods on the available datasets averaged over 10 random runs.

size learning rate of 0.1, the smoothing coefficient was selected to be 0.5, and the shrinkage coefficient to be 0.99.

We also compared our approach to the publicly available math exam solver RR2015 (Roy and Roth, 2015). This method is based on a combination of template-based features and categorizers. The accuracy performance was measured by counting the number of times the equation generated the correct results, so that 10 + 7 and 7 + 10 would both be considered to be correct. Results are shown on Table 5.

We can see that there is a large difference in performance between RR2015 and the RNN-based encoder-decoder approach. While their method seems to be very good on some datasets, it fails on CC, which is the dataset in which one needs two equations involving parentheses. On average, the trend is the following: using data only does not succeed in giving good results, and we can see that with generated data we are performing better already. This could be explained by the fact that the generators' vocabulary has a good overlap with the vocabulary of the real data. However, mixing real and generated data improves performance significantly. When GENERE is used, the sampling is tuned to the problem at hand and give better generalization performance.

To understand if GENERE learned a meaningful data generator, we inspected the coefficients $\gamma_1, \cdots, \gamma_8$ that are used to select the 8 data generators described earlier. This is shown is Figure 2.

The results are quite surprising at first sight: the AI2 dataset only involves additions and subtractions, but GENERE selects the generator generating divisions as the most important. Investigating, we noted that problems generated by the division generator were reusing some lexical items that were present in AI2, making the vocabulary very close to the problems in AI2, even if it does not cover division. We can also note that the differences in proportions are quite small among the 4 symbols $+, -, *$ and $/$ across all the datasets. We can also clearly see that the noisy generator 'N' and 'P' are not very relevant in general. We explain this by the fact that the noise induced by these generators is too artificial to generate relevant data for training. Their likelihood on the model trained on real data remains small.

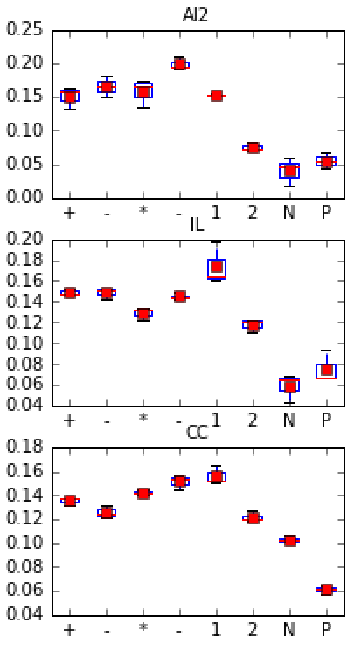

Figure 2: Base generators proportions learned by GENERE.

4 Conclusion

In this work, we argued that many problems can be solved by high-capacity discriminative probabilistic models, such as deep neural nets, at the expense of a large amount of required training data. Unlike the current trend which is to reduce the size of the model, or to define features well targeted for the task, we showed that we can completely decouple the choice of the model and the design of a data generator. We proposed to allow data generators to be "weakly" specified, leaving the undetermined coefficients to be learned from data. We derived an efficient algorithm called GENERE, that jointly estimates the parameters of the model and the undetermined sampling coefficients, removing the need for costly cross-validation. While this procedure could be viewed as a generic way of building informative priors, it does not rely on a complex integration procedure such as Bayesian optimization, but corresponds to a simple modification of the standard stochastic optimization algorithms, where the sampling alternates between the use of real and generated data. While the general framework assumes that the sampling distribution is differentiable with respect to its learnable parameters, we proposed a Gaussian integration trick that does not require the data generator to be differentiable, enabling practitioners to use *any* data sampling code, as long as the generated data resembles the real data.

We also showed in the experiments, that a simple way to parametrize a data generator is to use a mixture of base generators, that might have been derived independently. The GENERE algorithm learns automatically the relative weights of these base generators, while optimizing the original model. While the experiments only focused on sequence-to-sequence decoding, our preliminary experiments with other high-capacity deep neural nets seem promising.

Another future work direction is to derive efficient mechanisms to guide the humans that are creating the data generation programs. Indeed, there is a lack of generic methodology to understand where to start and which training data to use as inspiration to create generators that generalize well to unseen data.

Acknowledgments

We would like to thank Subhro Roy for helping us run his model on our new data splits. We are very thankful to Thomas Demeester, Johannes Welbl and Matko Bošnjak for their valuable feedback. Lastly, we would like to thank the three anonymous reviewers for their helpful comments and feedback.

This work was supported by a Marie Curie Career Integration Award and an Allen Distinguished Investigator Award.

References

Guillaume Bouchard and Bill Triggs. 2004. The trade-off between generative and discriminative classifiers. In *16th IASC International Symposium on Computational Statistics (COMPSTAT'04)*, pages 721–728.

Kyunghyun Cho, Bart Van Merriënboer, Çalar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October. Association for Computational Linguistics.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. *arXiv preprint arXiv:1601.01280*.

Albert Gatt and Ehud Reiter. 2009. Simplenlg: A realisation engine for practical applications. In *Proceedings of the 12th European Workshop on Natural Language Generation*, pages 90–93. Association for Computational Linguistics.

Mohammad Javad Hosseini, Hannaneh Hajishirzi, Oren Etzioni, and Nate Kushman. 2014. Learning to solve arithmetic word problems with verb categorization. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 523–533, Doha, Qatar, October. Association for Computational Linguistics.

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

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

P. Niyogi, F. Girosi, and T. Poggio. 1998. Incorporating prior information in machine learning by creating virtual examples. *Proceedings of the IEEE*, 86(11):2196–2209, Nov.

Sinno Jialin Pan and Qiang Yang. 2010. A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on*, 22(10):1345–1359.

Tim Rocktäschel, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, and Phil Blunsom. 2016. Reasoning about entailment with neural attention. In *International Conference on Learning Representations*.

Subhro Roy and Dan Roth. 2015. Solving general arithmetic word problems. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1743–1752. Association for Computational Linguistics.

Subhro Roy, Tim Vieira, and Dan Roth. 2015. Reasoning about quantities in natural language. *Transactions of the Association for Computational Linguistics*, 3:1–13.

Bernhard Scholkopf and Alexander J Smola. 2001. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey E. Hinton. 2014. Grammar as a foreign language. *CoRR*, abs/1412.7449.

Ronald J Williams. 1988. On the use of backpropagation in associative reinforcement learning. In *Neural Networks, 1988., IEEE International Conference on*, pages 263–270. IEEE.