Edit Embedding via Reinforcement Learning

Shunfu Mao
Electrical Engineering
shunfu@uw.edu

Joshua Fan
Computer Science
jyf2@uw.edu

1 Problem Statement

Edit embedding has been a research topic of interest in theoretical computer science [2]. The **edit distance** (or Levenshtein distance) between strings x and y is defined to be the minimum number of edit operations (insertion, deletion and substitution) to transform string x into y. Edit distance is an important metric for many applications, including spell checking, correction systems for optical character recognition, and clustering genetic sequences. Unfortunately, computing edit distance for long strings is computationally expensive, since best known exact algorithms for computing edit distance run in nearly quadratic time (in the length of the strings) [1]. Thus, it is generally infeasible to compute the edit distance between every pair of strings in a large dataset, even though that would be very useful for various clustering applications.

Instead, we would like to learn a transformation f which maps one string (s) to another ($a = f(s)$), such that for any pair of sequences (s_i, s_j), the Hamming distance between the transformed strings ($d_H(f(s_i), f(s_j))$ is close to the true edit distance between the original strings ($d_e(s_i, s_j)$). To do this, we aim to minimize the following non-differentiable [1] loss function:

$$L = \frac{1}{|P|} \sum_{(s_i,s_j)\in P} ||\frac{d_H(f(s_i), f(s_j))}{d_e(s_i, s_j)} - 1||^2 \tag{1}$$

2 Model

In this section, we describe our models for the edit embedding problem.

2.1 Sequence-to-Sequence (Seq2Seq) Architecture

Existing edit embedding algorithms designed by computer scientists try to find a mapping f that transforms input sequence s into the output sequence a. A recent one is called CGK embedding [2], which constructs a from s by repeating bits of s in a randomized way. The algorithm is briefly described as follows, and is illustrated in Fig 1

- initialize index $i = 0$ for $s \in S^{L_s}$ and $j = 0$ for $a \in S^{3L_s}$, and a random binary matrix $R \in \{0, 1\}^{|S|\times 3L_s}$. S is the alphabet set, such as binary $\{0, 1\}$ or DNA $\{A, T, C, G\}$.
- update $a[j]$ by $a[j] \leftarrow s[i]$
- update i by $i \leftarrow R[s[i]][j]$ and j by $j \leftarrow j + 1$

Since the CGK embedding already provides a (randomized) function to transform arbitrary strings into new strings such that the Hamming distance between the new strings is a good approximation of the edit distance between the original strings, we can start off by training a neural network in a supervised way to learn a function that imitates the CGK embedding. By first training the neural network to imitate a theoretically-sound edit embedding approach, we hope to provide a good

[1] We need to do thresholding in order to obtain a from s.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

i	1	2	3	4	5	6	7	8	9	10	
x	A	C	G	T	G	A	C	G	T	G	
	i	i+R[j][f(x)[j]]									
R[j][f(x)[j]]	1	2	3	4	5	6	7	8	9	10	... 30
A	0	1	0	0	1	0	1	1	0	1	...
C	1	1	0	1	1	1	1	0	0	0	...
G	0	1	1	1	0	0	0	1	1	1	...
T	1	0	0	0	1	0	1	1	0	1	...
j	1	2	3	4	5	6	7	8	9	10	... 30
f(x)	A	A	C	C	G	G	G	G	T	T	...
	j	j=j+1									

Figure 1: A CGK example where input/output consist of DNA alphabets. When i equals 1, the source $x[i] = A$ is copied to the target $f(x)[j], j = 1$. Since $R[j][f(x)[j]] = 0$, we proceed $j = j + 1 = 2$ but keeps i. So next $f(x)[j], j = 2$ is still A, copied from $x[i], i = 1$. Now the rand bit $R[j][f(x)[j]] = R[2][A] = 1$, we proceed $j = j + 1 = 3$ and $i = i + 1 = 2$. So next $f(x)[j], j = 3$ is C, copied from $x[i = 2]$

initialization for the model before applying reinforcement learning. We naturally consider applying recurrent neural network related architectures for sequence problems. In particular, we consider f to be a sequence-to-sequence (Seq2Seq) architecture (based on e.g. LSTM [5]) that's been widely used in machine translation [14][3], as illustrated in Fig 3.

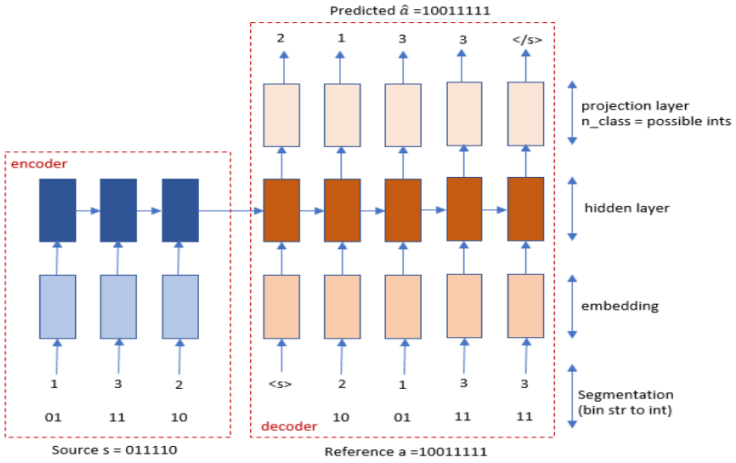


Figure 2: Seq2Seq architecture. The encoder network maps an arbitrary-length sequence to a fixed-length embedding vector, and the decoder network decodes that embedding into a new sequence. In our scenario, we segment our sequence s into blocks and view them as the input words. We generate output words corresponding to the CGK embedding.

2.2 Siamese Network Architecture (SNA)

The CGK embedding may not be the optimal transformation to minimize the loss; we may be able to learn a better transformation. To approximate $d_e(s_1, s_2)$ by $d_H(a_1, a_2)$, we consider training a Siamese network architecture (SNA) [8][9] with two copies of f for minimizing a certain loss, as illustrated in Fig 3. Note such an architecture may or may not contain a decoder[2]. We think the model with a decoder is more reasonable because the input sequence lengths can vary.

For each training example, we simultaneously run two different strings forward through a shared network to obtain embeddings for each string. Then, the loss can be calculated as a function of both embeddings, and errors backpropagate through a shared network. In our situation, the loss function L in Eq. 1 is non-differentiable, because the Hamming and Edit distances between strings are discrete-valued. Thus, a straightforward supervised approach using standard backpropagation shall not work. However, reinforcement learning algorithms (Sec 3) such as the REINFORCE policy

[2]If there's no decoder, the loss is based on the input sequences s_{i1}, s_{i2}, and their embedding vectors v_{i1}, v_{i2}

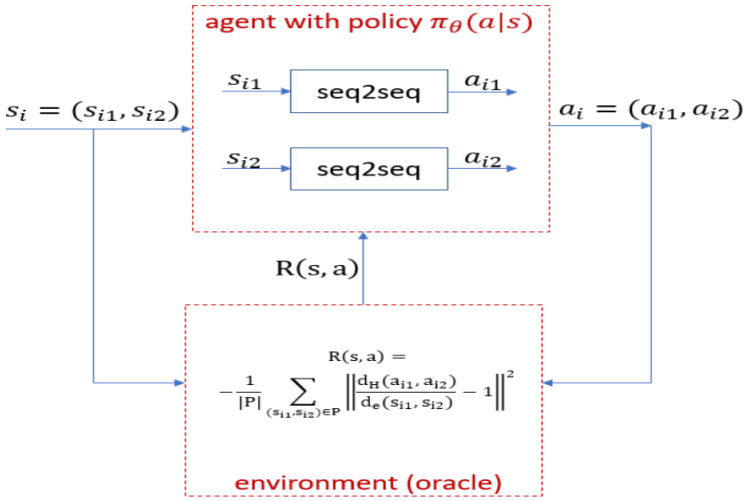


Figure 3: Reinforcement Learning via a Siamese Network Architecture (SNA)

gradient algorithm) allow us to train networks that maximize a reward, even when the reward function is non-differentiable. In particular, we can treat the SMA as **agent**, the pair of input sequences (s_i, s_j) as **state**, the output embedding (a_i, a_j) (non-thresholded, softmax) as **actions**, and $-L$ as **rewards**. These methods have been recently explored in dialogue generation [7], image captioning [12], text summarization [10] and question answering [15].

3 Reinforcement Learning (RL) Algorithms

3.1 Monte Carlo REINFORCE

The REINFORCE algorithm directly differentiates the policy $\pi(\hat{a}_t|s_t;\theta)$ with respect to the policy's parameters θ. In our problem, our actions do not influence future states, so we can drop the time-step t. The expected reward of any particular time step can be written as

$$\eta(\theta) = \mathbb{E}_{\hat{a}\sim\pi(a|s,\theta)}[R(s,\hat{a})]$$

where $R(s,\hat{a})$ is the reward computed based on the edit distance between the input sequences $s = (s_i, s_j)$, and the Hamming distance between the transformed sequences $\hat{a} = (\hat{a}_i, \hat{a}_j)$. $\pi(\hat{a}|s,\theta)$ is the probability of producing action $\hat{a}$ from the neural network parameterized by θ.

The gradient of the expected reward can be simplified to

$$\nabla_\theta \eta(\theta) = \mathbb{E}_{\hat{a}\sim\pi(\hat{a}|s,\theta)}[R(s,\hat{a})\nabla_\theta \log \pi(\hat{a}|s,\theta)]$$

We can approximate this expectation by simply sampling a single action $\hat{a}$ from the policy. Now we can adjust the policy parameters in the direction in which the expected reward will increase using policy gradient. This gives us the following algorithm (based off Monte Carlo REINFORCE).

- Sample K states (pairs of input sequences), $s_1 \dots s_K$.
- From each state s_k, sample T actions (pairs of output sequences), $\hat{a}_{k1}, \dots \hat{a}_{kT}$ from the current policy $\pi(\hat{a}|s_k,\theta)$. For simplicity, we may use T=1.
- The gradient is calculated by:

$$\nabla_\theta \eta(\theta) \approx \frac{1}{K}\frac{1}{T}\sum_{t=1}^{T}\sum_{i=1}^{K}[R(s_k,\hat{a}_{kt})\nabla_\theta \log \pi(a_{kt}|s_k,\theta)]$$

- Update the network parameters by:

$$\theta \leftarrow \theta + \alpha\nabla_\theta \eta(\theta)$$

3.1.1 Self-critical training

The expected gradient computed using mini-batches under the REINFORCE algorithm usually has very high variance. Thus, there have been many attempts to normalize the reward by some baseline, typically an estimate of future rewards. For this problem, we adopt the Self-Critical Sequence Training approach, where we normalize our rewards by the rewards observed by the test-time system. At test time, the decoder greedily selects the best action at each time step, whereas during training, the decoder samples actions from the probability distribution induced by the model. If the proposed sampled action receives higher reward than the current action, we shift the model's parameters to output that action more frequently.

For the above equations, we replace $R(s_k, \hat{a}_{kt})$ with $R(s_k, \hat{a}_{kt}) - R(s_k, \hat{a}^*)$, where $R(s_i, \hat{a}^*)$ is the reward obtained by the model using the test time inference algorithm (greedy decoding), and $R(s_i, \hat{a}_{kt})$ is the reward obtained by the proposed action that was sampled from the network.

3.2 Policy Gradients with Parameter-Based Exploration (PGPE)

As suggested, we've surveyed another method called Policy Gradients with Parameter-Based Exploration (PGPE). It's firstly proposed in [13] in order to sample directly in parameter space for lower-variance gradient estimates. In [4] it's claimed to have better performance compared with evolution strategies (e.g. ES and CMA-ES algorithms) on training a Multi-Dimentional Recurrent Neural Network for a simpler version of Go game.

To understand PGPE, let's consider our original goal is to find parameter θ (e.g. SNA parameter) that maximizes $J(\theta) = \int_H p(h|\theta) r(h) dh$ by calculating $\nabla_\theta J(\theta)$ (h represents an episode and $r(h)$ is related reward) and updating θ via gradient ascent.

Instead of calculating θ directly, PGPE introduces another parameter ρ that determines the distribution of θ. Consequently the problem now becomes to find ρ that maximizes $J(\rho) = \int_\Theta \int_H p(h, \theta|\rho) r(h) dh d\theta$ by calculating $\nabla_\rho J(\rho)$ and updating ρ via gradient ascent. It can be shown that

$$\nabla_\rho J(\rho) \approx \frac{1}{N} \Sigma_{n=1}^{N} \nabla_\rho \log p(\theta|\rho) r(h^n) \tag{2}$$

Here ρ is updated every N episodes. In our actual implementation, we consider one training batch as an episode. Typically PGPE assumes $\rho = \{\rho_i = (\mu_i, \sigma_i)\}$ and each $\theta_i^{(n)}$ - the i-th parameter at episode n of θ has normal distribution with mean μ_i and deviation σ_i. By this assumption, we can directly calculate $\nabla_\rho \log p(\theta|\rho)$ in Equ 2 by

$$\nabla_{\mu_i} \log p(\theta|\rho) = \frac{\theta_i^{(n)} - \mu_i}{\sigma_i^2} \tag{3}$$

$$\nabla_{\sigma_i} \log p(\theta|\rho) = \frac{(\theta_i^{(n)} - \mu_i)^2 - \sigma_i^2}{\sigma_i^3} \tag{4}$$

This implies we can update ρ first via gradient ascent and then sample agent (network) parameter θ, without doing backpropagation of neural network. In our actual implementation, we also include a baseline, as suggested in Algorithm 1 in [13]. The procedure is described as follows:

- Initialize $\rho = \{(\mu_i, \sigma_i)\}$. We initialize μ_i based on pretrained θ_i and σ_i from uniform distribution of a small range.
- Sample N episodes (one episode corresponds to one training batch). For each n-th episode, we sample $\theta^{(n)}$ from ρ where each $\theta_i^{(n)} \in \theta$ has normal distribution with mean μ_i and deviation σ_i. Based on $\theta^{(n)}$ of this episode, we calculate $\nabla_\rho \log p(\theta|\rho)$ explicitly using Equ 3, Equ 4; We also obtain from the training batch the input states $s_1 \dots s_K$ and corresponding actions $a_1 \dots a_K$, and we calculate rewards $r(h^n)$ from them.
- Note: (1) We maintain a smoothed baseline b where b is updated per batch by $b \leftarrow 0.9b + 0.1r(h^n)$. We then apply $r(h^n) \leftarrow r(h^n) - b$ before updating meta param ρ. (2) We've tried several $r(h^n)$ functions as will be described in more detail in Section 4.4.

- By collecting $\nabla_\rho J(\rho)$ and $r(h^n)$ from N episodes, we then update μ_i and σ_i via gradient ascent using Equation 2.
- Repeat till convergence of objective function (e.g. $J(\rho)$)

4 Evaluation

4.1 Data Preparation

We generate synthetic binary data for train and evaluation.

For the Seq2Seq model, each training or evaluation sample is a pair of (s_i, a_i) where i is the sample index, s_i represents a binary input sequence and a_i represents the CGK transform of s_i (also a binary sequence). In particular, the CGK generation is briefly described in Section 2.1 and is illustrated in Fig 1.

For the RL model where the agent is a Siamese Network, each training or evaluation sample is a triplet of $(s_{i1}, s_{i2}, d_e(s_{i1}, s_{i2}))$. Binary sequences s_{i1} and s_{i2} can be independent (e.g. dissimilar pair) or dependent (e.g. similar pair where s_{i2} is generated by passing s_{i1} through an indel channel with small insertion, deletion and substitution error rate). $d_e(s_{i1}, s_{i2})$ represents the edit distance between s_{i1}, s_{i2}.

4.2 Seq2Seq Evaluation

To evaluate the Seq2Seq model, we first train the model using $10K$ pairs of $\{(s_i, a_i)\}$ where $a_i = f_{CGK}(s_i)$. Source sequence s_i has length 50 and target sequence a_i, due to randomness of CGK embedding, has variable length ranging up to 150. As illustrated by Fig 3, the input sequences are segmented into blocks of length 5 and each block (e.g. a word) is represented by an integer id, and is translated into an embedding vector of size 100. The training aims to minimize the cross-entropy loss between predicted sequences $\{\hat{a}_i\}$ and reference sequences $\{a_i\}$. We update Seq2Seq parameters using Adam optimizer with learning rate 0.001, at every batch with batch size 100. The training lasts 5 epochs.

Every time we update Seq2Seq parameters during training, we also evaluate performance using $2K$ pairs of samples. Notice that the main difference between training and evaluation is that at training, decoder takes reference a_i as input while at evaluation the decoder takes previous hidden layer output e.g. $\hat{a}_i[t-1]$ as input for the prediction of $\hat{a}_i[t]$. The metric for evaluation mainly checks the Hamming distance between predicted sequence $\hat{a}_i$ and reference sequence a_i.

Fig 4 shows the initial evaluation of the performance trend, based on 1 layer LSTM for both encoder and decoder. In the first subplot (x-axis represents batch iterations and y-axis represents objective loss values), the training cross entropy steadily decreases. The training Hamming loss decreases from 60 to around 40, meaning the predicted CGK embedding $\hat{a}_i = f(s_i)$ has Hamming distance of about 40 from the true CGK embedding a_i. When the embedding length is about 100 to 150, this is about $26\% - 40\%$ mismatches. The validation Hamming loss is very high initially and reaches around 50 later, meaning about $33\% - 50\%$ mismatches. The second and third subplots indicate that as training proceeds, the average predicted length of $\hat{a}_i$ converges to the target length of a.

We also improve the model by using BiLSTM and attention mechanism. This further reduces the training cross entropy loss to around 20. However, the validation Hamming loss is not reduced. We also tuned number of epochs, embedding size and block length, they are helpful to reduce entropy loss further (e.g. cross entropy loss around 1) but not helpful to decrease validation Hamming loss. This suggests that Seq2Seq model does not seem to generalize to learn CGK embedding well, however as we can see next, it is helpful to separate similar and dis-similar input sequence pairs.

The Seq2Seq model is useful as a pretrain step, before we train SNA using RL. To see this, Fig 5 illustrates the SNA performance without reinforcement learning. The left subplot is the histogram of Hamming distance between CGK transformed pairs. The orange curve is where original input pairs are similar (as described in Sec 4.1 and thus the Hamming distance tends to be small, whereas the blue curve is where original input pairs are dis-similar and thus the Hamming distance tends to be large. It is clear that these two groups are well seperated. The right subplot in Fig 5 is the histogram

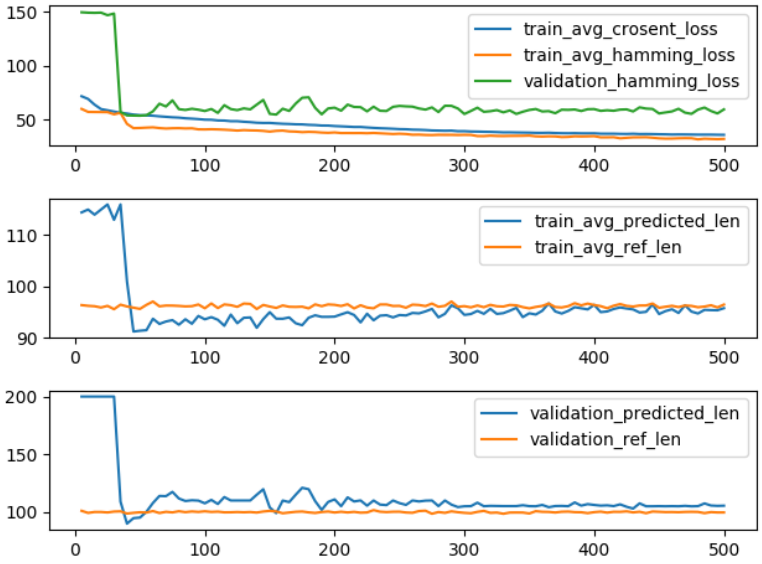
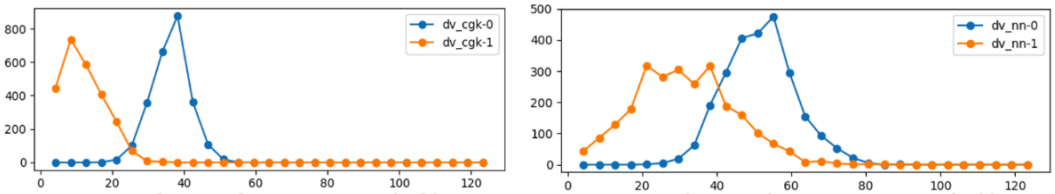


Figure 4: Initial evaluation of Seq2Seq architecture



Hamming distances between CGK embeddings (blue is independent pairs; orange is similar)

Hamming distances between Seq2Seq embeddings (blue is independent; orange is similar)

Figure 5: Histogram of deviation of hamming distance between predicted sequences from edit distance between input sequences

of Hamming distance between Seq2Seq transformed pairs. The Seq2Seq learned models separates the similar and dis-similar input sequence pairs to some extent, but not as well as CGK.

We also check the histogram of $R(s,a) = d_H(\hat{a}_{i1}, \hat{a}_{i2})/d_e(s_{i1}, s_{i2}) - 1$, the deviation of the transformed Hamming distance from the original edit distance, for both CGK and our SNA. But the related orange and blue curves are not well-separated. So this may not be a good metric to intuitively see the clustering effect. This motivates us to try additional reward functions for RL models.

4.3 REINFORCE Evaluation

We used the REINFORCE algorithm to attempt to improve the pre-trained policy. We plot the results (reward over time) of a run in figure 6.

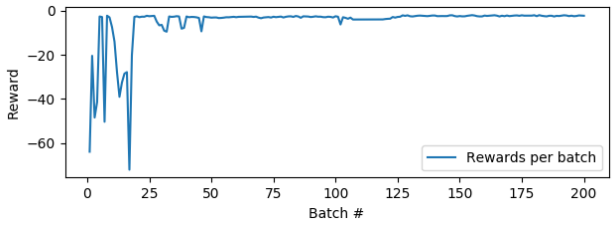


Figure 6: Reward $(-(\frac{d_H(a_1,a_2)}{d_e(s_1,s_2)} - 2)^2)$ over time

The reward does improve to some extent, but the training is unstable and there is a lot of noise. For example, the results are highly sensitive to the model's initialization and random choices. Sometimes, the model outputs the same sequence for every single input, which is a common failure mode for LSTMs. Even when it avoids that, the algorithm generally converges to a poor policy, as shown in figure 7. The rewards incurred in the end are around -2 to -2.5, which does not do a great job of distinguishing between similar versus different sequence pairs:

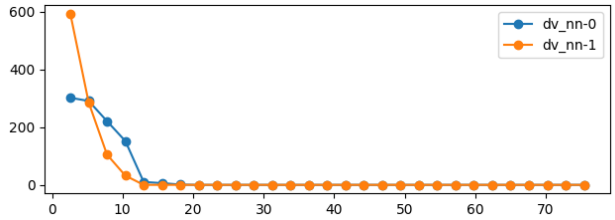


Figure 7: Histogram of Hamming distance between predicted sequences for independent pairs (blue) and similar pairs (orange)

As shown in figure 7, the learned policy does not do a good job of distinguishing similar sequence pairs from different sequence pairs. If the input sequences are different, the model is somewhat more likely to result in a high Hamming distance between the transformed sequences, but it is not guaranteed.

4.3.1 Modifications to Reward Function

Unfortunately the reward function $-(\frac{d_H(a_1,a_2)}{d_e(s_1,s_2)} - 2)^2$ appears to be difficult to fully optimize. There are extremely few policies that achieve high enough reward to approximate the edit distance with reasonable accuracy. We tried modifying the reward function to explicitly favor separation between similar pairs and independent pairs in the transformed sequences, as follows:

$$R(s,a) = \begin{cases} -|d_H(a_1,a_2) - d_e(s_1,s_2)| & \text{if sequences are similar} \\ d_H(a_1,a_2) & \text{if sequences are independent} \end{cases}$$

If the sequences are similar (low edit distance), the agent is encouraged to output strings whose Hamming distance is similar to the edit distance. Otherwise, if the sequences are independent, the agent is encouraged to output strings that are as dis-similar as possible. This is also a difficult reward function to train on, as the agent is rewarded very differently depending on whether the input sequences were similar or different. The Hamming distances between the transformed sequences are plotted in figure 8

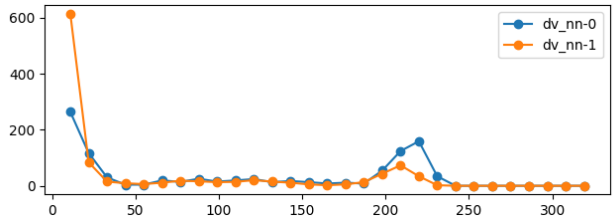


Figure 8: Histogram of Hamming distance between predicted sequences for independent pairs (blue) and similar pairs (orange)

The algorithm is more likely to map similar input sequences to similar transformed sequences, and different input sequences to different transformed sequences, which is a plus. However, there is still a high degree of noise, and the resulting embedding does not approximate edit distance at all. Also, the training was again unstable and most runs produced worse results.

4.4 PGPE Evaluation

To evaluate PGPE, we mainly train the RL framework and check how PGPE's rewards vary and how applying PGPE affects the clustering of similar and dis-similar input pairs.

To train PGPE, we initialize the meta-parameter $\rho = \{\rho_i = (\mu_i, \sigma_i)\}$ by initialize μ_i with pretrained θ_i and initialize σ_i with uniform distribution with range $[0, 0.01]$. If μ_i is randomly initialized or σ_i range is large, the similar and dis-similar pairs (e.g. the orange and blue curves in Fig 5) will not be distinguished. In addition, a large σ_i range or a large learning rate will make PGPE training unstable since negative σ_i will occur[3]

Unfortunately, PGPE does not show an improving reward (e.g. as defined in Fig 3) or a better separation between similar/dis-similar input pairs. To remedy this, we've tried to apply two other reward functions. One is per sample reward $R_2 = 1_{similar} \times (-d_H) + 1_{dis-similar} d_H$ which encourages small hamming distance between similar pairs and penalizes large hamming distance among dis-similar pairs. Fig 9 shows how applying PGPE affect this reward over iterations. As we can see, the reward of using PGPE is only slightly better than turning PGPE off. And it does not seem to improve the cluster separation. The other reward we tried is calculated per batch, $R_3 = \frac{1}{|P_{dis-similar}|} \sum_{s_i \in P_{dis-similar}} d_H(a_i) - \frac{1}{|P_{similar}|} \sum_{s_i \in P_{similar}} d_H(a_i)$ - we hope to enlarge the length between two clusters. The performance is similar as R_2.

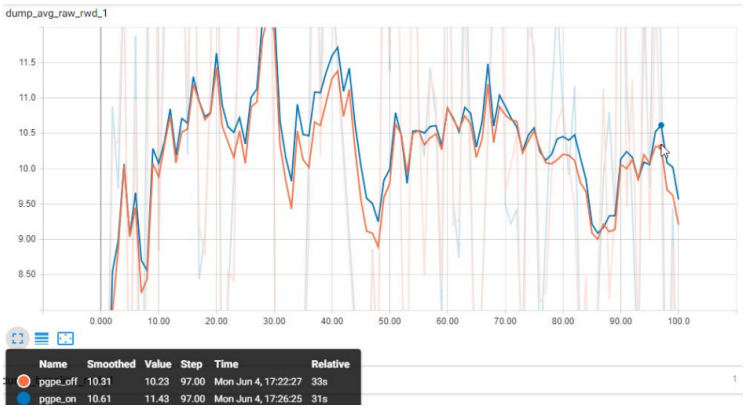


Figure 9: PGPE effect on reward R_2. x-axis represents episode iteration and y-axis is average reward R_2 per batch, with a smoothing factor 0.9

We speculate that perhaps there are too many parameters of the SNA model, so we also tried to reduce size of network parameter θ by using smaller blocklen, embedding size, or LSTM cell size. This hurts Seq2Seq performance at the very beginning and does not bring a separation between clusters.

5 Summary and Future Work

We attempted to apply reinforcement learning along with a Siamese Seq2Seq neural network architecture to the difficult problem of learning an edit embedding such that the Hamming distance between two embeddings approximates the edit distance between the original sequencesWhile the model performs better than chance, it does worse than the pre-existing CGK embedding in terms of distinguishing similar sequences from different ones. We were also unable to get reinforcement learning algorithms to produce a policy that produced effective edit embeddings. It was still a very interesting approach to try though, and we list a few suggestions for future work.

First, it may be interesting to try Natural Policy Gradient [6], which can be helpful in escaping from local optima. However, inverting the Fisher Matrix is computationally expensive when the number of parameters is large. Secondly, it is known that generating sequences using reinforcement learning is difficult due to the enormous action space. We could try to mitigate this through incremental learning and the MIXER algorithm [11]; the approach is to start by training a model based on cross-entropy loss (using the ground-truth CGK embedding as inputs to the decoder), and then gradually allow the decoder to make use of its own predictions when decoding. Finally, we could also try running our algorithms on larger and different datasets, especially real DNA datasets where we may be able to exploit structure in the data (instead of having sequences sampled uniformly at random).

[3]We flip σ_i when it contains negative values. Another suggested method is to try to update $\sigma_i' = log\sigma_i$ first and sample θ_i from $N(\mu_i, exp(\sigma_i'))$ to avoid the negative value issue.

References

[1] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless seth is false). In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*, STOC '15, pages 51–58, New York, NY, USA, 2015. ACM.

[2] Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 712–725, New York, NY, USA, 2016. ACM.

[3] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation, 2014.

[4] Mandy Grüttner, Frank Sehnke, Tom Schaul, and Jürgen Schmidhuber. Multi-dimensional deep memory atari-go players for parameter exploring policy gradients. In Konstantinos Diamantaras, Wlodek Duch, and Lazaros S. Iliadis, editors, *Artificial Neural Networks – ICANN 2010*, pages 114–123, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[6] Sham M Kakade. A natural policy gradient. In *Advances in neural information processing systems*, pages 1531–1538, 2002.

[7] Jiwei Li, Will Monroe, Alan Ritter, Michel Galley, Jianfeng Gao, and Dan Jurafsky. Deep reinforcement learning for dialogue generation, 2016.

[8] Jonas Mueller and Aditya Thyagarajan. Siamese recurrent architectures for learning sentence similarity. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2786–2792. AAAI Press, 2016.

[9] Paul Neculoiu, Maarten Versteegh, and Mihai Rotaru. Learning text similarity with siamese recurrent networks. In *Proceedings of the 1st Workshop on Representation Learning for NLP*. Association for Computational Linguistics, 2016.

[10] Romain Paulus, Caiming Xiong, and Richard Socher. A deep reinforced model for abstractive summarization, 2017.

[11] Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. Sequence level training with recurrent neural networks. *CoRR*, abs/1511.06732, 2015.

[12] Steven J. Rennie, Etienne Marcheret, Youssef Mroueh, Jarret Ross, and Vaibhava Goel. Self-critical sequence training for image captioning, 2016.

[13] Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Policy gradients with parameter-based exploration for control. In Véra Kůrková, Roman Neruda, and Jan Koutník, editors, *Artificial Neural Networks - ICANN 2008*, pages 387–396, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014.

[15] Caiming Xiong, Victor Zhong, and Richard Socher. Dcn+: Mixed objective and deep residual coattention for question answering, 2017.