Generate and Revise: Reinforcement Learning in Neural Poetry

A Preprint

Andrea Zugarini[*]
DINFO, DIISM
University of Florence, University of Siena
Florence 50139 Italy, Siena, 53100 Italy
andrea.zugarini@unifi.it

Luca Pasqualini
DIISM
University of Siena
Siena 53100 Italy
pasqualini@diism.unisi.it

Stefano Melacci
DIISM
University of Siena
Siena 53100 Italy
mela@diism.unisi.it

Marco Maggini
DIISM
University of Siena
Siena 53100 Italy
maggini@diism.unisi.it

February 9, 2021

Abstract

Writers, poets, singers usually do not create their compositions in just one breath. Text is revisited, adjusted, modified, rephrased, even multiple times, in order to better convey meanings, emotions and feelings that the author wants to express. Amongst the noble written arts, Poetry is probably the one that needs to be elaborated the most, since the composition has to formally respect predefined meter and rhyming schemes. In this paper, we propose a framework to generate poems that are repeatedly revisited and corrected, as humans do, in order to improve their overall quality. We frame the problem of revising poems in the context of Reinforcement Learning and, in particular, using Proximal Policy Optimization. Our model generates poems from scratch and it learns to progressively adjust the generated text in order to match a target criterion. We evaluate this approach in the case of matching a rhyming scheme, without having any information on which words are responsible of creating rhymes and on how to coherently alter the poem words. The proposed framework is general and, with an appropriate reward shaping, it can be applied to other text generation problems.

1 Introduction

Developing machines that reproduce artistic behaviours and learn to be creative is a long-standing goal of the scientific community in the context of Artificial Intelligence [1, 2]. Recently, several researches focused on the case of the noble art of Poetry, motivated by success of Deep Learning approaches to Natural Language Processing (NLP) and, more specifically, to Natural Language Generation [3, 4, 5, 6, 7, 8]. However, existing Machine Learning-based poem generators do not model the natural way poems are created by humans, i.e., poets usually do not create their compositions all in one breath. Usually a poet revisits, rephrases, adjusts a poetry many times, before reaching a text that perfectly conveys their intended meanings and emotions. In particular, a typical feature of poems is that the composition has also to formally respect predefined meter and rhyming schemes.

With the aim of developing an artificial agent that learns to mimic this behaviour, we design a framework to generate poems that are repeatedly revisited and corrected, in order to improve the overall quality of the poem. We frame this problem as a navigation task approached with Reinforcement Learning (RL), exploiting Proximal Policy Optimization (PPO) [9] that, to our best knowledge, is not commonly applied to Natural Language Generation, despite being an

[*] Corresponding author: http://sailab.diism.unisi.it/people/andrea-zugarini/

improved instance of the more common Vanilla Policy Gradient (VPG). In the task of generating and progressively editing the draft of a poem until it matches a target rhyming scheme, we show that PPO leads to better results than VPG. The agent is not informed about what a rhyme is and how to implement the considered scheme, making the task extremely challenging in an RL perspective. The agent generates a draft poem and it corrects the draft one word at a time. It not only understands that the ending words of each verse are the ones that are important with respect to the rhyming scheme, but that also other words of the poem might need to be adjusted to make the poem coherent with the rhyming words. Despite the application to poetry generation, the proposed framework is general and it can be applied to other text generation problems, provided an opportune reward shaping.

This paper is organized as follows. After discussing related work (Section 1.1), the neural models are described in Section 2, while the RL-based poem revision dynamics is detailed in Section 3. Experiments are reported in Section 4 and, finally, conclusions are drawn in Section 5.

1.1 Related Work

Early methods on Poetry Generation [10] addressed the problem with rule-based techniques, whereas more recent approaches focused on learnable neural language models. The first deep learning solutions tackled Chinese Poetry. In [3], authors combined convolutional and recurrent networks to generate quatrains. Afterwards, both [5] and [4] proposed a sequence-to-sequence model with attention mechanisms. In the context of English Poetry, transducers were exploited to generate poetic text [6]. The generation structure (meter and rhyme) is learned from characters by cascading a module considering the context, with a weighted state transducer. Recently, in Deep-speare [7], the authors generated English quatrains with a combination of three neural models that share the same character-based embeddings. One network is a character-aware language model predicting at word level, another neural model learns the meter, and the last one identifies rhyming pairs. Generated quatrains are finally selected after a post-processing step from the output of the three modules. In [8], the authors focused on a single Italian poet, Dante Alighieri, by making use of a syllable-based language model, that was trained with a multi-stage procedure on non-poetic works of the same author and on a large Italian corpus.

Reinforcement Learning has been recently used in several Natural Language Generation applications, such as Text Summarization [11, 12, 13], Machine Translation [14] and Poem Generation [15, 16] as well. However, most of the proposed approaches exploit RL as a mean to make common evaluation metrics differentiable, such as BLEU and ROUGE scores [17]. Of course, these metrics can be computed only in those tasks in which the target text (ground truth) is available. In [15] the authors extended Generative Adversarial Networks (GANs) [18] to the generation of sequences of symbols, through Reinforcement Learning. The GAN discriminator is used as a reward signal for a RL-based language generator, and, among a variety of tasks, their framework was applied to Chinese quatrains generation. In [16], a mutual Reinforcement Learning scheme was used to improve the quality of the generated Chinese quatrains. In both works, different generic rewards were designed exploiting the simplest policy-based RL algorithm, i.e. Vanilla Policy Gradient. Surprisingly, Proximal Policy Optimization is less commonly used in the scope of Natural Language Generation, despite leading to a more robust and efficient RL algorithm [19].

Our generate-and-revise framework is related to retrieve-and-edit seq2seq approaches [20, 21, 22, 23, 24], where text generation reduces to an adaptation/paraphrasing of the retrieved template(s) related to the current input. The refinement process can be optimized with standard seq2seq learning algorithms because of the presence of revised targets. In our generate-and-revise instead, we neither start from retrieved templates, nor we have reference revisions. That is why we cast the problem as a navigation task and exploit RL to learn a revision policy that adjusts draft poems in order to improve their quality.

2 Generate and Revise Poems

Our framework is rooted on the idea that creating a poem is a multi-step process. First, the draft of a new poem is generated. Then, an iterative revision procedure is activated, in which the initial draft is progressively edited. We model this problem by means of a *generator*, that creates the draft, and a *reviser*, that edits the draft up to the final version of the poem. The reviser is structured as an iterative procedure that, at each iteration, identifies a word of the poem which does not suit well the context in which it is located, and substitutes it with a better word. At each step the reviser has to decide both *which* word to replace and *with what*. A straightforward approach to implement this idea is to design an RL agent that jointly addresses both the tasks. Thus, given an m-word poem with vocabulary size $|V|$, the agent has to choose among a large number of actions, i.e. $|V| \cdot m$, due to usually large $|V|$ (in the order of tens of thousands in our experiments). Therefore the problem quickly becomes extremely hard to tackle.

We keep the idea of exploiting an RL-based approach, but we decouple the problem implementing the reviser with two learnable models, namely the *detector* and the *prompter*, each of them responsible of one of the two aforementioned tasks, i.e., detecting a word to substitute (detector), and suggesting how to change a target word (prompter), respectively. The *generator*, the *detector*, and the *prompter* are based on neural architectures, trained from scratch with appropriate criteria, while the *detector* is fully developed by means of RL. The whole scheme is sketched in Fig. 1. The structure of this module allows us to reduce the action space of the RL procedure to (up to) N words in the poem, making it independent on $|V|$. The prompter identifies the words in V that are most compatible with the surrounding context.

In the following, the generator (Section 2.1), the detector (Section 2.2) and the prompter (Section 2.3) will be described in detail, whereas the RL-dynamics of the detector are presented in Section 3.

2.1 Conditional Poem Generator

The poem generation procedure is an instance of Natural Language Generation based on a learnable Language Model (LM). Before considering the specific details of Poetry, we describe the LM used in this work. Let us consider a sequence of tokens $(w_1, \ldots, w_{n+m})$ taken from a text corpus in a target language. For convenience in the description, let us divide the tokens into two sequences $\boldsymbol{x}$ and $\boldsymbol{y}$, where $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{y} = (y_1, \ldots, y_m)$. The former ($\boldsymbol{x}$) is the context provided to the text generator from which to start the production of new text. More generally, $\boldsymbol{x}$ is a source of information that conditions the generation of $\boldsymbol{y}$ ($\boldsymbol{x}$ could also be empty). The goal of the LM is to estimate the probability $p(\boldsymbol{y})$, that is factorized as follows,

$$p(\boldsymbol{y}) = \prod_{i=1}^{m} p(y_i | y_{<i}, \boldsymbol{x}), \tag{1}$$

being $y_{<i}$ a compact notation to indicate the words in the left context of y_i. Notice that when the sequence $\boldsymbol{x}$ has size $n = 0$, we fall back to the traditional LM formulation [25]. The text generation is the outcome of sampling the next sequence $\boldsymbol{y}$ from (1). Machine Translation, Text Summarization, Text Continuation, Poem Generation, and in general any sequence-to-sequence problem in NLP can be formulated as in (1). The way $p(y_i|y_{<i}, \boldsymbol{x})$ will be related to the input sequence $\boldsymbol{x}$ depends on how strongly $\boldsymbol{x}$ is informative with respect to $\boldsymbol{y}$. Problems in which the source sequence significantly biases the generation outcome are referred as *non-open-ended* text generation, in contrast to *open-ended* text generation, where the source sequence loosely correlates with the output $\boldsymbol{y}$ [26].

Poem Generation is an instance of *open-ended* text generation. When starting to generate a novel poem from scratch, there is, of course, no source input sequence. After having generated a few verses or when starting from a few given verses, the next-verses generation can be conditioned using them (i.e., $\boldsymbol{x}$ contains previously given verses, while $\boldsymbol{y}$ is about the verses to be generated), but there is still a huge degree of freedom in the possible verses that can be generated, due to the intrinsically creative nature of Poetry. There might be several features to further constrain the LM with information that does not come from the input text, and, in this work, we consider two important features, that are the author a and the target rhyme scheme r. We update (1) by introducing the information on a and r,

$$p(\boldsymbol{y}) = \prod_{i=1}^{m} p(y_i | y_{<i}, \boldsymbol{x}, a, r). \tag{2}$$

that is the reference equation on which our poem generation is based.

We model the distribution in Equation 2 by means of a sequence-to-sequence neural architecture with attention. Our LM is a variant of [27], similar to the one proposed in [7], and it is based on an encoding-decoding scheme. The encoder is responsible of creating a compact representation of $\boldsymbol{x}$, while the decoder yields a probability distribution over the words in V given the outcome of the encoder, and the conditioning signals a and r leading to $p(y_i|y_{<i}, \boldsymbol{x}, a, r)$.

Encoding. The encoder of $\boldsymbol{x}$ computes a contextual representation of each word x_j of the input sequence $\boldsymbol{x}$ (n words), by means of a bidirectional LSTM (bi-LSTM). The output of this module is the set $H_x = \{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_n\}$, being $\boldsymbol{h}_j$ the contextualized representation of the j-th word. In detail, at each time step j, the bi-LSTM is fed with the concatenation of the word embedding $\boldsymbol{w}_j \in \mathbb{R}^d$ associated to x_j, and $\boldsymbol{u}_j \in \mathbb{R}^r$, a character-based representation of x_j. We indicate with $\overrightarrow{\boldsymbol{h}}_j$, $\overleftarrow{\boldsymbol{h}}_j$ the internal states of the bi-LSTM processing the sequence of augmented word representations,

$$\overrightarrow{\boldsymbol{h}}_j = \overrightarrow{LSTM}_{\text{encx}}([\boldsymbol{w}_j, \boldsymbol{u}_j], \overrightarrow{\boldsymbol{h}}_{j-1}),$$
$$\overleftarrow{\boldsymbol{h}}_j = \overleftarrow{LSTM}_{\text{encx}}([\boldsymbol{w}_j, \boldsymbol{u}_j], \overleftarrow{\boldsymbol{h}}_{j+1}),$$

where $\overrightarrow{LSTM}$, $\overleftarrow{LSTM}$ are the functions computed by the LSTMs in the two directions. The final representation of the j-th word of the input sequence is $\boldsymbol{h}_j = [\overrightarrow{\boldsymbol{h}}_j, \overleftarrow{\boldsymbol{h}}_j]$. Overall, the encoder outputs $H_x = \{\boldsymbol{h}_1, \ldots, \boldsymbol{h}_n\}$. The char-based representation $\boldsymbol{u}_j$ is obtained by processing the word characters with another bi-LSTM. We augment $\boldsymbol{w}_j$ with a char-based representation to better encode sub-word information, that is crucial to capture rhyming schemes and meter in the poems.

Decoding. The decoder is responsible of returning the distribution $p(y|y_{<i}, \boldsymbol{x}, a, r)$ at each time index i, and, when used to generate text, to sample a word from p.

We stack two recurrent layers. First an LSTM that computes at each time step i a representation $\boldsymbol{z}_i$ given the previous word y_{i-1} merged with author (a) and rhyme scheme (r) information encoded in form of embeddings $\boldsymbol{a}$ and $\boldsymbol{r}$, obtaining:

$$\boldsymbol{z}_i = \mathrm{LSTM}_{\mathrm{dec}}([\boldsymbol{w}_{i-1}, \boldsymbol{u}_{i-1}, \boldsymbol{a}, \boldsymbol{r}], \boldsymbol{z}_{i-1}),$$

where $\boldsymbol{w}_{i-1}$ is the word embedding of y_{i-1} and $\boldsymbol{u}_{i-1}$ is the character-aware word representation shared with the encoder. Thanks to the inputs $\boldsymbol{a}$ and $\boldsymbol{r}$, the state $\boldsymbol{z}_i$ includes author-specific and rhyme-scheme-specific information. This allows the system to generate text that is oriented toward the given author style and the target rhyme scheme. The second recurrent layer is a Gated Recurrent Unit (GRU) cell [28] that progressively fuses $\boldsymbol{z}_i$ with the context data in H_x, in order to create a further vector $\boldsymbol{q}_i \in \mathbb{R}^d$ that compactly includes all the conditioning signals of (2). First, an attention mechanism [27] is applied over the encoding of the words of $\boldsymbol{x}$, i.e, on their contextualized representations collected in H_x, yielding an attention-based representation $\boldsymbol{c}_i$ of $\boldsymbol{x}$,

$$\boldsymbol{c}_i = \mathrm{attn}(\boldsymbol{z}_i, H_x).$$

Then, the concatenation of $\boldsymbol{c}_i$ with the representation $\boldsymbol{z}_i$ of the triple $(y_{<i}, a, r)$ is processed by a GRU cell,

$$\boldsymbol{q}_i = \mathrm{GRU}([\boldsymbol{c}_i, \boldsymbol{z}_i], \boldsymbol{z}_{i-1}),$$

Finally, the distribution $p(y|y_{<i}, \boldsymbol{x}, a, r)$ is obtained through a linear projection of $\boldsymbol{q}_i$ with the transposed embedding matrix $E' \in \mathbb{R}^{d \times |V|}$, and then applying the softmax function. The model is trained to maximize $p(\boldsymbol{y})$ on a text corpora of poems (see Section 4).

Generation. Poems are generated sampling from p. As a matter of fact, the sampling strategy plays a crucial role in the quality of the generated text, and it has been recently shown to have a major impact in Natural Language Generation [29]. We preferred nucleus (top-p) sampling, with $p = 0.9$, to generate quatrains over multinomial and top-k sampling. We indicate with $\boldsymbol{o}$ the sequence of words sampled from p that will consitute a draft poem. The drafts poems generated by the model will be then revised by the joint work of detector and prompter modules.

2.2 Detector

Once we have generated a draft poem using the model of Section 2.1, a detection module learns to select the next word of the draft that needs to be revised. The detector is a neural model that yields a probability distribution $\pi(o_i|\boldsymbol{o}, a, r)$ over the N words of the poem. Of course, in order to detect which words to replace, it is important to take into account the author and rhyme information. In detail, the words of poem $\boldsymbol{o}$ are encoded by a network that is analogous to the encoder of $\boldsymbol{x}$ in Section 2.1. The word representations, collected in H_o, are processed by an attention mechanisms attn_{det}, building a compact embedding of the whole poem that is also function of the author and of the rhyme scheme. Then, a Multi-Layer Perceptron (MLP) with softmax activation in the output layer returns the probability over the N words,

$$\pi(o_j|\boldsymbol{o}, a, r) = \mathrm{MLP}_j(\mathrm{attn}_{det}([\boldsymbol{a}, \boldsymbol{r}], H_o)) \tag{3}$$

being MLP_j the j-th output unit. Multinomial sampling applied to π leads to the selection of the word(s) that should be replaced. This module is trained by RL, as we will describe in Section 3.

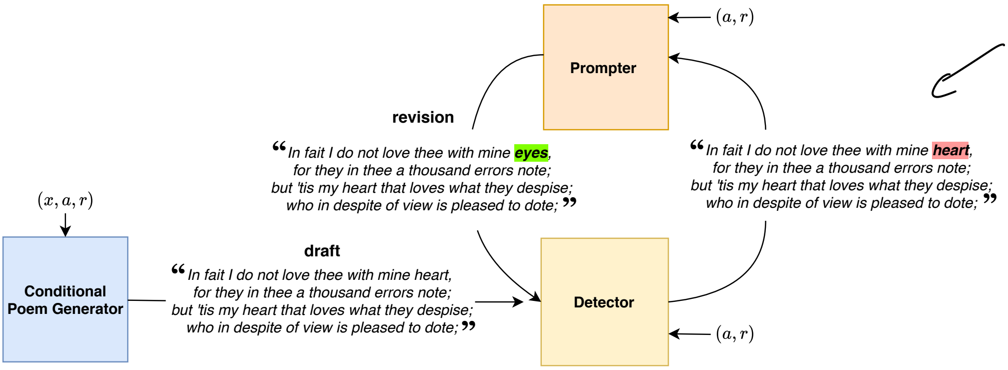
2.3 Prompter

The role of the prompter module is to provide valid candidates to replace the word previously selected by the detector of Section 2.2. The prompter module solves the problem of modeling language given the left-right contexts of each word, that can be formulated following an approach similar to the one exploited by the conditional LM of (2). Thus, given an author a and a rhyme scheme r, we use a neural model to learn the following distribution from data,

$$p(\boldsymbol{o}) = \prod_{i=1}^{N} p(o_i|o_{<i}, o_{>i}, a, r), \tag{4}$$

being $o_{<i}$, $o_{>i}$ the words in left and right context of o_i, respectively. Once p($\boldsymbol{o}$) has been learnt, we can sample $p(o_i|o_{<i}, o_{>i}, a, r)$ to get one or more candidate words for replacing the selected one.

The prompter network follows the context encoding schemes of [30] and [31]. In particular, the words of poem $\boldsymbol{o}$ are encoded by a network that computes representations of the left and right contexts around each target word, discarding the target word itself. Differently from the encoding of $\boldsymbol{o}$ in Section 2.2, here the final representation of the j-th word is then $[\overrightarrow{\boldsymbol{h}}_{j-1}, \overleftarrow{\boldsymbol{h}}_{j+1}]$.[2] This representation is concatenated with the author embedding $\boldsymbol{a}$ and the rhyme scheme embedding $\boldsymbol{r}$, followed by a learnable linear layer with softmax activation that projects the concatenated vector to the space of vocabulary indices. Including $\boldsymbol{a}$ and $\boldsymbol{r}$ in the prompter module is crucial in order to allow the network to learn how to revise a target word in function of the poet and rhyme scheme. Candidate(s) for replacing the selected word are sampled from $p(o_i|o_{<i}, o_{>i}, a, r)$, as discussed in the Poem Generator of Section 2.1. In this case we used top-k sampling ($k = 50$) to have a large pool of candidates. The prompter is trained to maximize $p(\boldsymbol{o})$ on a text corpora of poems (Section 4).



“In fait I do not love thee with mine **eyes**,
for they in thee a thousand errors note;
but 'tis my heart that loves what they despise;
who in despite of view is pleased to dote;”

“In fait I do not love thee with mine **heart**,
for they in thee a thousand errors note;
but 'tis my heart that loves what they despise;
who in despite of view is pleased to dote;”

“In fait I do not love thee with mine heart,
for they in thee a thousand errors note;
but 'tis my heart that loves what they despise;
who in despite of view is pleased to dote;”

Figure 1: Overall Generate and Revise scheme on an example poem. The conditional poem generator (light blue module) produces a draft poem, which is iteratively revised by the detector (pale yellow) - Prompter (light orange) modules until satisfaction of a certain criteria. At each step the detector identifies the word to replace, *heart* highlighted in red, while the prompter is responsible for finding the substitute, *eyes* highlighted in green.

Beyond Narrative Description
- provides a dataset for Free Verse
- We want to Implement a way

Implement Constraints
Either ryhme schemes (ex: ABBA) or Form (ex: Haiku)
- Form requires more altering

How do we get constraints for Haiku?
- Count Syllables
- Count lines

3 Revision as a Navigation Task

Once the poem generator and the prompter modules have been trained, the task of revising a generated poem consists in detecting which words to change and letting the prompter replace them. If we assume to change one word at a time, we can easily consider this task as a decision process in the space of the dictionary words V. Each decision defines which word to change at that a given step, and the prompter replaces it with a suitable candidate. The sequence of decisions is the *policy* of an agent whose goal is to improve the text, according to a given reward function. Text revision stops when a satisfying score has been reached. This task may be cast as a navigation problem, where the current *state* of the agent is identified by the sequence of the words in the current text revision. This allows us to reformulate the problem as an RL task where the navigation space is the environment [32], while the decisions are identified by actions executed by the agent in the environment. We provide a brief introduction of Reinforcement Learning in Appendix A.

A RL task can be framed as a sequential decision-making problem in which, at each step t, the agent observes a state $S_t \in \mathcal{S}$ from the environment, and then selects an action $A_t \in \mathcal{A}$. The environment yields a numerical reward $R_{t+1} \in \mathcal{R}$ and then it moves to the next state S_{t+1}. This interaction gives raise to a *trajectory* of random variables. In our task, since words are elements of the vocabulary V, we have that $\mathcal{S}$ is the space of the poems of length N with words from V for the target author a and with rhyming scheme r, $\mathcal{A}$ is the set of indices of the word positions in the poem plus the do-nothing action, while $\mathcal{R} \subset \mathbb{R}$. To define a reward function we use the shortest path problem formulation. The agent aims at reaching the final text revision in the least amount of steps. Conventionally this means that the reward R_t is defined as a negative number for each state not at the goal state position and a positive number

[2] In our implementation, we used the same LSTMs when encoding data in the detector and in the prompter module.

or zero when the goal state is reached. Formally, if $\boldsymbol{o}_t$ is the poem revision at step t, we have

$$\begin{aligned} A_t &= \hat{A}_t \in \bigcup_{g=1}^{N+1} \{g\} && (5) \\ S_{t+1} &= (\boldsymbol{o}_{t+1}, a, r) && (6) \\ R_{t+1} &= \begin{cases} 1 & \text{if } S_{t+1} = S_f \\ -1 & \text{otherwise} \end{cases} && (7) \end{aligned}$$

where S_f is the goal state in which the text is not revised anymore.

The natural connection between the modules presented in Section 2 and the RL-based setting is easily established once we redefine (3) as the probability of an action in the state described by the triple $(\boldsymbol{o}, a, r)$, that perfectly suits the definition in (6), yielding a *policy* function. Using Deep Neural Networks (DNNs) to approximate the RL-related functions, as we do in the case of the probability distribution over the action space π, is a pretty common approach in nowadays RL-based problems (see, e.g., [32]). In the following descriptions, we compactly rewrite (3) adding the symbol θ to refer to the network weights that are learned by means of the RL procedure, i.e., $\pi(\cdot|\cdot;\theta)$. Policy Gradient methods are suitable for navigation tasks, as shown in [33], especially when the states' space becomes large [34]. In such spaces often off-policy algorithms (like Q-Learning) are indeed observed to be unable to converge. In this work, we compare two on-policy RL algorithms: Vanilla Policy Gradient (Section 3.1) and Proximal Policy Optimization (Section 3.2).

3.1 Vanilla Policy Gradient

Vanilla Policy Gradient (VPG) [35] is an on-policy RL algorithm whose aim is to learn a policy without using q-values as a proxy. This is obtained increasing the probabilities of actions that lead to higher return, and decreasing the probabilities of actions that lead to lower return. Actions are usually sampled from a multinomial distribution for discrete actions' spaces and from a normal distribution for continuous action spaces. VPG works by updating policy parameters θ via stochastic gradient ascent on policy performance over a buffer built from a certain number of trajectories,

$$\theta \longleftarrow \theta + \alpha \nabla_\theta J(\pi(\cdot|\cdot;\theta))$$

where $J(\pi(\cdot|\cdot;\theta))$ denotes the expected finite-horizon undiscounted return of the policy and ∇_θ its gradient with respect to θ ($\alpha > 0$). In order to compute $J(\pi(\cdot|\cdot;\theta))$, the algorithm requires to evaluate further actions for each state s in the buffer. In this paper, we use Generalized Advantage Estimation (GAE) [36] to compute such actions, and the obtained rewards are saved and normalized with respect to "when" they are collected (the so called rewards-to-go). These are solutions reported in literature to be stable and to improve overall training performance of the model.

3.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) [9] is another on-policy RL algorithm which improves upon VPG. It is considered the state-of-the-art in policy optimization methods and it is a modified version of Trust Region Policy Optimization (TRPO) [37]. Both methods try to take the biggest possible improvement step on a policy using the currently available data, without stepping "too far" and making the performance collapse. This is done by maximizing a surrogate objective, subject to a constraint on policy update quantity, where such constraint depends on the KL-divergence between the old policy and the new policy after the update. Specifically, PPO uses a clipped objective to heuristically constrain the KL-divergence,

$$\max_\theta \mathbb{E}[\min(\rho_t \bar{A}_t, \text{clip}(\rho_t, 1-\epsilon, 1+\epsilon) \cdot \bar{A}_t),$$

where $\rho_t = \frac{\pi(A_t|S_t;\theta_t)}{\pi_{\theta_{old}}(A_t|S_t;\theta_{t-1})}$ is a policy ratio, clip$(\rho_t, \cdot, \cdot)$ clips ρ_t in the interval defined by the last two arguments, ϵ is an hyperparameter (we set $\epsilon = 0.2$), $\bar{A}_t$ is the estimated advantage function at time step t and A_t, S_t are respectively the action and the state at time step t. In the implementation used in this paper, when parameters θ are updated over a buffer of trajectories, the update process is early stopped if the constraint is not respected, thus avoiding the new policy to step "too far" from the previous one.

4 Experiments

We collected poems in English language from the Project Gutenberg[3] using the GutenTag tool [38] to filter out non-poetic work and collections. We also discarded non-English contents that occasionally appeared in the retrieved

[3] https://www.gutenberg.org/

documents. Poems are organized in stanzas, according to their XML-based description. Each stanza was then divided into quatrains, if not already in such format, and we assigned a rhyming scheme to each stanza, from a fixed dictionary of rhyming schemes. Rhymes were automatically detected with the Pronouncing library[4] and a few additional heuristic rules to cover most of the undetected rhymes. Long poems without any rhyming pattern were discarded as well. We used the meta-information about the author to define the authorship of the stanza, when available. We considered the most 768 frequent authors, the rest was marked as unknown. Overall, we obtained $757,891$ quatrains, divided in three sets of the sizes $684,100$, $36,006$ and $37,785$, respectively, used to train, validate and test the models. We limited the word vocabulary to the most frequent $50,000$ words, assigning an embedding of size 300 for all the models in all the experiments. The maximum sequence length of a quatrain has been set to 50, longer verses were truncated.

We define multiple experimental settings and tasks in order to evaluate the quality of the each module proposed in this work, up to the entire system that includes all the modules and the full pipeline of generation and iterative revision.

4.1 Conditional Poem Generator

While the proposed generator of Section 2.1, follows an established neural architecture, the innovative elements we introduce in this work are about the poem-related conditional features, *author* and *rhyme scheme*, and their use in Poetry with character-aware representations. We considered the task of generating a quatrain $\boldsymbol{y}$ given the context sequence $\boldsymbol{x}$ that is the previous quatrain, where the rhyme scheme is a symbol indicating the rhymes of an eight-verse poem. We considered the 50 most frequent rhyme schemes of size eight. The architecture hyper-parameters were commonly selected by choosing the best configuration on the validation set for the vanilla (i.e., not conditioned by author and rhyme) generator. The bi-LSTM state encoding the context sequence $\boldsymbol{x}$ was set to 512, as the state of the decoder $LSTM_{enct}$, and the GRU cell as well. Author and rhyme embedding sizes were set to 128 and 256, respectively.

We compared in terms of generation perplexity a model trained with or without any of the newly introduced conditional features, reporting results in Table 1. The conditional features allows the LM to be more accurate, that is an important result considering the open-ended challenging nature of the poem generation task.

4.2 Prompter

A similar analysis was followed to evaluate the quality of the prompter model of Section 2.3. In particular, we trained a prompter model on single quatrains, enforcing it to learn how to predict a word given its context. We used a bi-LSTM state of 1024 units. Again, the role of the new conditional features is what we are mostly interested in and, observing results of Table 2, we can see that they improve the suggestion quality. This result is in line with the case of Section 4.1, confirming the importance of further poem-related information.

Table 1: Perplexity measured on the validation (Val) and test (Test) sets of the poem generator, trained with or without conditional features.

	Val	Test
Vanilla Generator	52.98	59.78
Conditional Generator	**51.40**	**54.86**

Table 2: Perplexity measured on the validation (Val) and test (Test) sets of the prompter module, trained with or without conditional features.

	Val	Test
Vanilla Prompter	14.09	14.78
Conditional Prompter	**12.90**	**13.40**

4.3 Revision as a Navigation Task

In order to show the quality of the detector module and that approaching text correction as shortest path problem is feasible, we created "corrupted" poems from real poems in the dataset by replacing one or more words in random positions with words sampled from the entire vocabulary V.[5] The agent operates in an environment where each episode starts with a corrupted poem, and it has to learn to reconstruct the original not-corrupted poem, selecting at

[4] `https://pypi.org/project/pronouncing/`

[5] Frequent words are sampled as replacement more often than rare ones.

each step which word to change. In this artificial setting we assume that, once the agent picks which word to substitute, a perfect prompter (oracle) will replace it with the ground truth, i.e. the word originally positioned there in the real poem. This means that after each agent action, the selected position will be either replaced with the original word, in case of a corrupted word, or nothing will be changed, in case of a correct word. The navigation terminates when the goal state is reached, that occurs after all the corrupted words are removed from the poem.

The MLP predicting actions has a single hidden layer of size 512. We performed different experiments over this poem reconstruction environment, using a PPO-based agent. Each experiment differs in the number of poems that the agent has to fix, and the number of words perturbed in the poem. We considered $\{1, 10, 100\}$ poems that, at the beginning of each episode, are randomly "corrupted" by altering 1 or 3 (referred to as "multiple") words in the original poem.Please note that even in the simplest case, the experiment with one poem only and a single perturbed word, the number of generated "corrupted" poems is huge, $|V|^{|x|}$, where $|x|$ is the poem length.

The PPO-agent is trained for 10 volleys in each experiment, with $1,000$/$20,000$/$200,000$ episodes in the experiments with 1/10/100 poems, respectively. We set the maximum episode length to 10 steps. Hence, the reward varies between $[-10, 1]$ where 1 corresponds to the case in which the agent immediately identifies the "corrupted" word (when there is only 1 corrupted word), and -10 indicates a full failure. Results are shown in Table 3. The "Volley 0" column defines the average total reward at the end of the first volley, while the "Volley 9" column defines such value at the end of the last volley. The reward value improves during the training volleys, while increasing the number of poems makes the problem exponentially more complex. Due to the dynamic perturbation of the poems on each volley, the policy learned by the agent is not tied to a set of perturbed words, but it is general over the set of poems the agent is trained on. However, independently from the amount of poems, results confirm that the revision problem can be framed into a shortest path problem and addressed by RL using PPO, in a varying amount of time.

Table 3: Results of the experiments with the PPO-based agent on poem reconstruction task of Section 4.3. The averaged total rewards after the first volley and the last volley are reported, respectively.

	R Volley 0	R Volley 9
1 Poem	-7.866	-0.425
1 Poem (multiple)	-9.092	-3.126
10 Poems	-7.793	0.293
100 Poems	-8.110	-6.389

Table 4: Two examples of generated poems with generate and revise approach given a target rhyme scheme, before and after the revision iterative steps.

Rhyme scheme	Draft	Revision
AABB	*the mist that made us sweat and* ***ache*** *with toil, from doing good or ill,* *the hour when we were led to play* *the children of the people's* ***brood,***	*the mist that made us sweat and* ***chill*** *with toil, from doing good or ill,* *the hour when we were led to play* *the children of the people's* ***way,***
ABBB	*and when, above, the winter's snow* *has risen in the wintry* ***sky*** *and leaves their path to cloud's decay,* *and life is spent, and life is drear,*	*and when, above, the winter's snow* *has risen in the wintry night* ***away*** *and leaves their path to cloud's decay,* *and life is spent, and life is drear* ***today***

4.4 Generate and Revise Poems

Now we consider the complete system in which all the modules are active as in Fig. 1. We focused on the task of generating poems and progressively revising them, in which the agent goal is to substitute words so that the poem matches a target rhyme scheme. Episodes begin with poems generated by the conditional generator. This task is significantly more challenging than the previously described ones, since there is no ground truth for generated poems, and words replacements are provided by the prompter model described in Section 2.3. Therefore, we let the model free to change any word in the quatrain, without restricting the agent actions to words at the end of each verse. Basically, the agent does not know that rhymes are related to the ending words of some verses, while the only information it receives is the reward (or penalty) signal that tells if the poem fulfills the target rhyming scheme or not.

We ran several experiments comparing PPO with VPG, varying in each experiment the number of poems to revise in the environment in $\{10, 100, 200, 500, 1,000\}$. We set to the number of training steps per volley at $10,000$ for the experiment with 10 poems, and we increase it to $100,000$ in the other experiments. Additionally, we considered another experiment, indicated as `dynamic`, in the most difficult scenario, i.e., where the environment spawns new, unseen, artificially generated quatrains at each episode. In such a case we report results of PPO only, because using VPG always resulted in a failure. An episode ends either when the target rhyme scheme is matched or after 30 steps, that corresponds to the maximum episode length. Therefore, the reward of an episode ranges in the interval $[-30, 1]$. Differently from our previous work [8], we do not carry out human evaluations, since rhyme matching can be quantitatively measured through the reward. Indeed, the reward is a direct way to assess the revised poem quality, because it is proportional to the number of steps needed for adjusting the target rhyme scheme. In particular, from Equation 7 we can observe that the number of revising steps in an episode (i.e. where reaching the goal state S_f) is equivalent to $|R_f| + 2$.

Results are presented in Table 5. We can see that, while the agent improves the reward in all the experiments with PPO, learning with VPG is not stable, and performs poorly. The superiority of PPO over VPG is also illustrated in Fig. 2, where we can see the instability of VPG in contrast to the steady progresses of PPO. Even if the task is very challenging, the model is able to strongly improve the average R score, thus indicating that it is actually moving the right steps in progressively fixing the rhymes. We also report in Table 4 two examples of draft revisions obtained with the agent trained with PPO in the `dynamic` environment.

Table 5: VPG vs PPO: Reward on the experiment of Section 4.4 with 10, 100, 200, 500 and 1000 poems. PPO is also evaluated with an environment that continuously generates new drafts (`dynamic`).

	N poems	R first Volley	R last Volley
VPG	10	-18.752	-14.630
PPO		-10.239	**-1.186**
VPG	100	-19.415	-19.264
PPO		-15.598	**-5.200**
VPG	200	-20.950	-18.432
PPO		-15.323	**-3.757**
VPG	500	-21.191	-19.623
PPO		-15.043	**-7.780**
VPG	1,000	-26.150	-21.179
PPO		-11.579	**-9.733**
PPO	`dynamic`	-14.796	-12.415

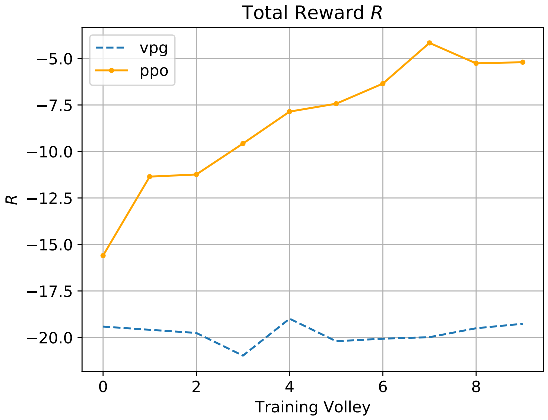


Figure 2: Rewards yielded by using PPO and VPG with respect to the number training volleys, in the experiment of Section 4.4 with 100 poems in the environment.

5 Conclusions and Future Work

In this paper we presented an innovative way of implementing the notion of creativity in a machine. Considering the task of automatically generating new poems, we proposed a model that implements the human-like behaviour of writing a draft and revising it multiple times. We proposed to create drafts that are conditioned to author and rhyme information, while the revision process is built around an iterative procedure that can be described as a navigation problem and solved with Reinforcement Learning with Proximal Policy Optimization, that significantly outperformed Vanilla Policy Gradient. Multiple experiments confirmed that the proposed approach is feasible and that it allows the machine to learn how to revise text, even if it is not explicitly instructed on which portion of text it should revise. The proposed framework is also general enough to be eventually applied to other text generation tasks, that is what we are going to do in future work.

References

[1] Margaret A. Boden. "Chapter 9 - Creativity". In: *Artificial Intelligence*. Ed. by Margaret A. Boden. Handbook of Perception and Cognition. San Diego: Academic Press, 1996, pp. 267–291. ISBN: 978-0-12-161964-0. DOI: `https://doi.org/10.1016/B978-012161964-0/50011-X`. URL: `http://www.sciencedirect.com/science/article/pii/B978012161964050011X`.

[2] Simon Colton, Geraint A Wiggins, et al. "Computational creativity: The final frontier?" In: *European Conference on Artificial Intelligence (ECAI)*. Vol. 12. Montpelier. 2012, pp. 21–26.

[3] Xingxing Zhang and Mirella Lapata. "Chinese poetry generation with recurrent neural networks". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 670–680.

[4] Qixin Wang et al. "Chinese song iambics generation with neural attention-based model". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. AAAI Press. 2016, pp. 2943–2949.

[5] Xiaoyuan Yi, Ruoyu Li, and Maosong Sun. "Generating chinese classical poems with rnn encoder-decoder". In: *Chinese Computational Linguistics and Natural Language Processing Based on Naturally Annotated Big Data*. Springer, 2017, pp. 211–223.

[6] Jack Hopkins and Douwe Kiela. "Automatically generating rhythmic verse with neural networks". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vol. 1. 2017, pp. 168–178.

[7] Jey Han Lau et al. "Deep-speare: A joint neural model of poetic language, meter and rhyme". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2018, pp. 1948–1958.

[8] Andrea Zugarini, Stefano Melacci, and Marco Maggini. "Neural Poetry: Learning to Generate Poems Using Syllables". In: *International Conference on Artificial Neural Networks*. Springer. 2019, pp. 313–325.

[9] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[10] Simon Colton, Jacob Goodwin, and Tony Veale. "Full-FACE Poetry Generation." In: *ICCC*. 2012, pp. 95–102.

[11] Romain Paulus, Caiming Xiong, and Richard Socher. "A deep reinforced model for abstractive summarization". In: *arXiv preprint arXiv:1705.04304* (2017).

[12] Yen-Chun Chen and Mohit Bansal. "Fast Abstractive Summarization with Reinforce-Selected Sentence Rewriting". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2018, pp. 675–686.

[13] Shashi Narayan, Shay B Cohen, and Mirella Lapata. "Ranking Sentences for Extractive Summarization with Reinforcement Learning". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. 2018, pp. 1747–1759.

[14] Luisa Bentivogli, Matteo Negri, Marco Turchi, et al. "Machine Translation for Machines: the Sentiment Classification Use Case". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 2019, pp. 1368–1374.

[15] Lantao Yu et al. "Seqgan: Sequence generative adversarial nets with policy gradient". In: *Thirty-First AAAI Conference on Artificial Intelligence*. 2017.

[16] Xiaoyuan Yi et al. "Automatic Poetry Generation with Mutual Reinforcement Learning". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 2018, pp. 3143–3153.

[17] Anja Belz and Ehud Reiter. "Comparing automatic and human evaluation of NLG systems". In: *11th Conference of the European Chapter of the Association for Computational Linguistics*. 2006.

[18] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[19] Yi-Lin Tuan et al. "Proximal Policy Optimization and its Dynamic Version for Sequence Generation". In: *arXiv preprint arXiv:1808.07982* (2018).

[20] Kelvin Guu et al. “Generating sentences by editing prototypes”. In: *Transactions of the Association for Computational Linguistics* 6 (2018), pp. 437–450.

[21] Ziqiang Cao et al. “Retrieve, rerank and rewrite: Soft template based neural summarization”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2018, pp. 152–161.

[22] Yuan Li et al. “Hybrid retrieval-generation reinforced agent for medical image report generation”. In: *Advances in neural information processing systems*. 2018, pp. 1530–1540.

[23] Jason Weston, Emily Dinan, and Alexander H Miller. “Retrieve and refine: Improved sequence generation models for dialogue”. In: *arXiv preprint arXiv:1808.04776* (2018).

[24] Nabil Hossain, Marjan Ghazvininejad, and Luke Zettlemoyer. “Simple and effective retrieve-edit-rerank text generation”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 2020, pp. 2532–2538.

[25] Yoshua Bengio et al. “A neural probabilistic language model”. In: *Journal of machine learning research* 3.Feb (2003), pp. 1137–1155.

[26] Ari Holtzman et al. “The curious case of neural text degeneration”. In: *arXiv preprint arXiv:1904.09751* (2019).

[27] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).

[28] Junyoung Chung et al. “Empirical evaluation of gated recurrent neural networks on sequence modeling”. In: *arXiv preprint arXiv:1412.3555* (2014).

[29] Abigail See et al. “Do Massively Pretrained Language Models Make Better Storytellers?” In: *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*. 2019, pp. 843–861.

[30] Oren Melamud, Jacob Goldberger, and Ido Dagan. “context2vec: Learning generic context embedding with bidirectional lstm”. In: *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*. 2016, pp. 51–61.

[31] Giuseppe Marra et al. “An unsupervised character-aware neural approach to word and context representation learning”. In: *International Conference on Artificial Neural Networks*. Springer. 2018, pp. 126–136.

[32] V Madhu Babu, U Vamshi Krishna, and SK Shahensha. “An autonomous path finding robot using Q-learning”. In: *2016 10th International Conference on Intelligent Systems and Control (ISCO)*. IEEE. 2016, pp. 1–6.

[33] Matt Knudson and Kagan Tumer. “Policy Search and Policy Gradient Methods for Autonomous Navigation”. In: *and Learning Agents Workshop at AAMAS 2010*. 2010.

[34] Luca Pasqualini and Maurizio Parton. “Pseudo Random Number Generation: a Reinforcement Learning approach”. In: *Procedia Computer Science* 170 (2020), pp. 1122–1127.

[35] Richard S Sutton et al. “Policy gradient methods for reinforcement learning with function approximation”. In: *Advances in neural information processing systems*. 2000, pp. 1057–1063.

[36] John Schulman et al. “High-dimensional continuous control using generalized advantage estimation”. In: *arXiv preprint arXiv:1506.02438* (2015).

[37] John Schulman et al. “Trust region policy optimization”. In: *International conference on machine learning*. 2015, pp. 1889–1897.

[38] Julian Brooke, Adam Hammond, and Graeme Hirst. “GutenTag: an NLP-driven tool for digital humanities research in the Project Gutenberg corpus”. In: *Proceedings of the Fourth Workshop on Computational Linguistics for Literature*. 2015, pp. 42–47.

[39] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018. ISBN: 9780262039246. URL: `https://books.google.it/books?id=6DKPtQEACAAJ`.

A Reinforcement Learning

Reinforcement Learning[6] (RL) is learning what to do in order to accumulate as much reward as possible during the course of actions. This very general description, known as *the RL problem*, can be framed as a sequential decision-making problem as follows.

Let us consider an *agent* interacting with an *environment*, through a set of possible *actions* that depends on the current situation, namely the *state*. An action affects the environment, therefore after each action the state will change. Some states are “better” than others, and the goodness of the state can be numerically quantifies with a value, called *reward*.

[6] For a comprehensive introduction to RL, see sections from 1.1 to 1.6 in [39].

The pair "state, reward" may possibly be drawn from a joint probability distribution, called the *model* or the *dynamics* of the environment. The agent will choose actions according to a certain strategy, called *policy* in the RL setting. The RL problem can then be stated as finding a policy maximizing the expected value of the total reward accumulated during the interaction agent-environment.

The RL problem implicitly assumes that the joint probability distribution of S_{t+1}, R_{t+1}, i.e. the state of the environment and the reward obtained at next time step $t+1$ depend only on the past via S_t and A_t, corresponding to the state of the environment and the action executed by the agent at time step t. In fact, the environment is fed only with the last action, and no other data from the history. This means that, for a fixed policy, the corresponding stochastic process $\{S_t\}$ is Markovian. When the agent experiences a trajectory, or episode, starting at time t, it accumulates a *discounted return* G_t:

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \qquad \gamma \in [0,1].$$

The return G_t is a random variable, whose probability distribution depends not only on the environment dynamics, but also on how the agent chooses actions in a certain state s. Choices of actions are encoded by the policy, i.e. a discrete probability distribution π on $\mathcal{A}$:

$$\pi(a|s) := \pi(a,s) := \Pr(A_t = a|S_t = s).$$

A discount factor $\gamma < 1$ is used mainly when rewards far in the future are less and less reliable or important, or in *continuing* tasks, that is, when the trajectories do not decompose naturally into *episodes*.

The average return from a state s, that is, the average total reward the agent can accumulate starting from s, represents how good is the state s for the agent *following the policy* π, and it is called *state-value* function:

$$v_\pi(s) := E_\pi[G_t|S_t = s].$$

Likewise, one can define the *action-value* function (known also as *quality* or *q-value*), encoding how good is *choosing an action* a *from* s *and then following the policy* π:

$$q_\pi(s,a) := E_\pi[G_t|S_t = s, A_t = a].$$

In most problems, like the one at hand, we have only a *partial knowledge of the environment dynamics*. This can be overcome by *sampling trajectories* $S_t = s, A_t = a, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, \ldots$. Policy Gradient (PG) algorithms estimate directly the policy $\pi(a|s;\theta)$ from sampled trajectories, without using a value function. The parameters vector θ_t at time t is often approximated by a neural network in a Deep Reinforcement Learning (DRL) fashion and it is modified to maximize a suitable scalar performance function $J(\theta)$, with the gradient ascent update rule:

$$\theta_{t+1} := \theta_t + \alpha \widehat{\nabla J(\theta_t)}.$$

Here the *learning rate* α is the step size of the gradient ascent algorithm, determining how much we are trying to improve the policy at each update, and $\widehat{\nabla J(\theta_t)}$ is any estimate of the performance gradient $\nabla J(\theta)$ of the policy. Different choices for the estimator corresponds to different PG algorithms. In this paper we use two PG algorithms, Vanilla Policy Gradient (Section 3.1) and Proximal Policy Optimization (Section 3.2).