Generating Text with Deep Reinforcement Learning

Hongyu Guo
National Research Council Canada
`hongyu.guo@nrc-cnrc.gc.ca`

Abstract

We introduce a novel schema for sequence to sequence learning with a Deep Q-Network (DQN), which decodes the output sequence *iteratively*. The aim here is to enable the decoder to first tackle easier portions of the sequences, and then turn to cope with difficult parts. Specifically, in each iteration, an encoder-decoder Long Short-Term Memory (LSTM) network is employed to, from the input sequence, automatically create features to represent the internal states of and formulate a list of potential actions for the DQN. Take rephrasing a natural sentence as an example. This list can contain ranked potential words. Next, the DQN learns to make decision on which action (e.g., word) will be selected from the list to modify the current decoded sequence. The newly modified output sequence is subsequently used as the input to the DQN for the next decoding iteration. In each iteration, we also bias the reinforcement learning's attention to explore sequence portions which are previously difficult to be decoded. For evaluation, the proposed strategy was trained to decode ten thousands natural sentences. Our experiments indicate that, when compared to a left-to-right greedy beam search LSTM decoder, the proposed method performed competitively well when decoding sentences from the training set, but significantly outperformed the baseline when decoding unseen sentences, in terms of BLEU score obtained.

1 Introduction

Many real-world problems can be effectively formulated as sequence to sequence learning. Important applications include speech recognition, machine translation, text rephrasing, question answering. For example, the last three can be expressed as mapping a sentence of words to another sequence of words. A major challenge of modeling these tasks is the variable length of sequences which is often not known a-priori. To address that, an encoder-decoder Long Short-Term Memory (LSTM) architecture has been recently shown to be very effective [8, 30]. The idea is to use one LSTM to encode the input sequence, resulting in a fixed dimensional vector representation. Subsequently, another LSTM is deployed to decode (generate) the output sequence, using the newly created vector as the LSTM's initial state. The decoding process is essentially a recurrent neural network language model [19, 29].

Decoding schema based on recurrent language models naturally fits a left-to-right decoding procedure, which aims to obtain an output sequence with the maximal probability or to select the top list of sequence candidates for further post-processing. In this paper, we propose an alternative strategy for training an end-to-end decoder. Specifically, we employ a Deep Q-Network (DQN) to embrace an iterative decoding strategy. In detail, the input sequence is first encoded using an encoder-decoder LSTM network. This process automatically generates both informative features to represent the internal states of and a list of potential actions for a DQN. Next, the DQN is employed to *iteratively* decode the output sequence. Consider rephrasing a natural sentence. This list of potential actions can contain the ranked word candidates. In this scenario, the DQN learns to make decision on which word will be selected from the list to modify the current decoded sequence. The newly edited output sequence is subsequently used as the input to the DQN for the next decoding iteration. Inspired by the recent success of attention mechanisms [3, 13, 20, 28], we here also bias the reinforcement

1

learning's attention, in each iteration, to explore sequence portions which are previously difficult to be decoded. The decoded sequence of the last iteration is used as the final output of the model. In this way, unlike the left-to-right decoding schema, the DQN is able to learn to first focus on the easier parts of the sequence, and the resulted new information is then use to help solve the difficult portions of the sequence. For example, a sentence from our testing data set was decoded by the encoder-decoder LSTMs as "Click here to read more *than* the New York Times .", which was successfully corrected by the DQN as "Click here to read more *from* the New York Times ." in the second iteration.

For evaluation, the proposed strategy was trained to encode and then decode ten thousands natural sentences. Our experimental studies indicate that the proposed method performed competitively well for decoding sentences from the training set, when compared to a left-to-right greedy beam search decoder with LSTMs, but significantly outperformed the baseline when decoding unseen sentences, in terms of BLEU [25] score obtained.

Under the context of reinforcement learning, decoding sequential text will need to overcome the challenge arise from the very large number of potential states and actions. This is mainly due to the flexible word ordering of a sentence and the existence of a large number of words and synonyms in modern dictionaries. To our best knowledge, our work is the first to decode text using DQN. In particular, we employ LSTMs to not only generalize informative features from text to represent the states of DQN, but also create a list of potential actions (e.g., word candidates) from the text for the DQN. Intuitively, the application of the DQN here also has the effect of generating *synthetic* sequential text for the training of the networks, because of the DQN's exploration strategy in training.

2 Background

Reinforcement Learning and Deep Q-Network Reinforcement Learning (RL) is a commonly used framework for learning control policies by a computer algorithm, the so-called agent, through interacting with its environment Ξ [1, 27]. Given a set of internal states $S = s_1, \ldots, s_I$ and a set of predefined actions $A = a_1, \ldots, a_k$, the agent takes action a at state s, by following certain policies or rules, will result in a new state s' and receive a reward r from Ξ. The aim of the agent is to maximize some cumulative reward through a sequence of actions. Each such action forms a transition tuple (s, a, r, s') of a Markov Decision Process (MDP). Practically, the environment is unknown or partially observed, and a sequence of state transition tuples can be used to formulated the environment.

Q-Learning [34] is a popular form of RL. This model-free technique is used to learn an optimal action-value function Q(s, a), a measure of the action's expected long-term reward, for the agent. Typically, Q-value function relies on all possible state-action pairs, which are often impractically to be obtained. A work around for this challenge is to approximate Q(s, a) using a parameterized function $Q(s, a; \theta)$. The parameter θ is often learned by features generalized over the states and actions of the environment [4, 31]. Promisingly, benefiting from the recent advance in deep learning techniques, which have shown be able to effectively generate informative features for a wide ranges of difficult problems, Mnih et al. [21] introduced the Deep Q-Network (DQN). The DQN approximates the Q-value function with a non-linear deep convolutional network, which also automatically creates useful features to represent the internal states of the RL.

In DQN, the agent interacts with environment Ξ in discrete iteration i, taking aim to maximize its long term reward. Starting from a random Q-function, the agent continuously updates its Q-values by taking actions and obtaining rewards, through consulting a current Q-value function. The iterative updates are derived from the Bellman equation, where the expectation E is often computed over all transition tuples that involved the agent taking action a in state s [31]:

$$Q_{i+1}(s, a) = E[r + \lambda \max_{a'} Q_i(s', a' | s, a) \tag{1}$$

Where λ is a discounted factor for future rewards.

DQN requires informative representation of internal states. For playing video games, one can infer state representations directly from raw pixels of screens using a convolutional network [21]. However, text sentences, for instance, not only contain sequential nature of text, but also have variable length. The LSTM's ability to learn on data with long range temporal dependencies and varying lengths makes it a natural choice to replace the convolutional network in the DQN for our application here. Next, we will briefly describe the LSTM network.

Long Short-Term Memory Recurrent Neural Networks Through deploying a recurrent hidden vector, Recurrent Neural Networks (RNNs) [1] can compute compositional vector representations for sequences of arbitrary length. The network learns complex temporal dynamics by mapping a length T input sequence $< x_1, x_2, \ldots, x_T >$ to a sequence of hidden states $< h_1, h_2, \ldots, h_T > (h_t \in R^N)$. The networks compute the hidden state vector via the recursive application of a transition function:

$$h_t = \Gamma(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \tag{2}$$

where Γ is an element-wise non-linearity sigmoid function; the W terms denote weight matrices (e.g. W_{xh} is the input-hidden weight matrix); b_h is hidden bias vector.

A popular variant of RNNs, namely LSTMs are designed to overcome the vanishing gradient issue in RNNs, thus better modeling long term dependencies in a sequence. In addition to a hidden unit h_t, LSTM includes input gate, forget gate, output gate and memory cell unit vectors, for the following purposes. The memory cell unit c_t, with a self-connection, is capable of considering two pieces of information. The first one is the previous memory cell unit c_{t-1}, which is modulated by the forget gate. Here, the forget gate embraces the hidden states to adaptively reset its cell unit through the self-connection. The second piece of information is a function of the current input and previous hidden state, modulated by the input gate. Intuitively, the LSTM can learn to selectively forget its previous memory or consider its current input. Similarly, the output gate learns how much of the memory cell to transfer to the hidden state. These additional cells enable the LSTM to preserve state over long periods of time [8, 12, 30, 32].
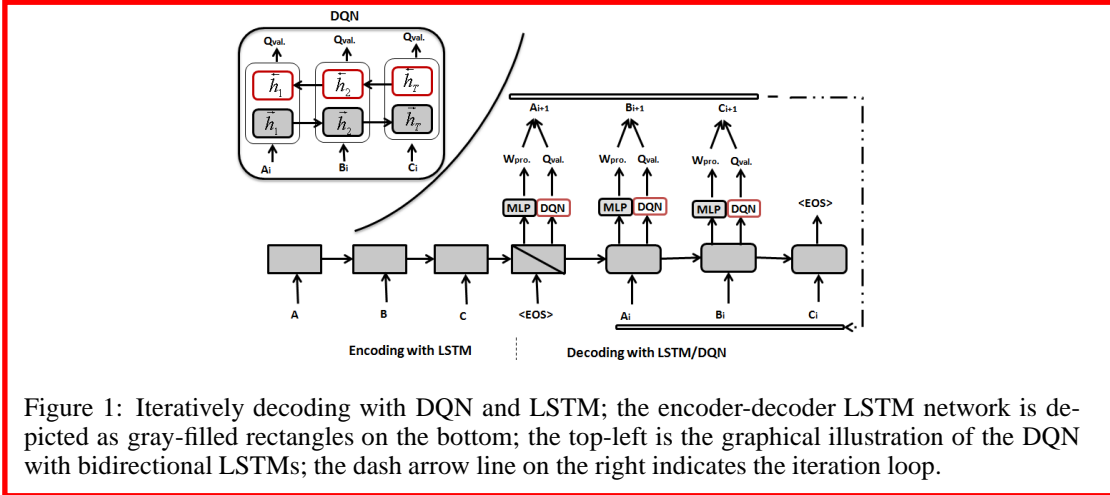

Figure 1: Iteratively decoding with DQN and LSTM; the encoder-decoder LSTM network is depicted as gray-filled rectangles on the bottom; the top-left is the graphical illustration of the DQN with bidirectional LSTMs; the dash arrow line on the right indicates the iteration loop.

3 Generating Sequence with Deep Q-Network

We employ an encoder-decoder LSTM network, as presented in [30], to automatically generate informative features for a DQN, so that the DQN can learn a Q-value function to approximate its long term rewards. The learning algorithm is depicted in Figure 1 and Algorithm 1.

3.1 Generating State Representations with LSTMs

The encoder-decoder LSTM network is depicted as gray-filled rectangles in Figure 1. For descriptive purpose, we named this State Generation Function (denoted as StateGF) under the context of DQN. In detail, given a natural sentence with N tokens, $< x_1, x_2, \ldots, x_N >$ (denoted as EnSen). We first encode the sequence using one LSTM (denoted as EnLSTM), reading into the tokens (e.g., words) one timestep at a time (e.g., $< A, B, C >$ in Figure 1). When reaching the end of the sentence (<EOS> in Figure 1), this encode process results in a fixed dimensional vector representation for the whole sentence, namely the hidden layer vector h_N^{en}. Next, the resulted h_N^{en} is used as the initial state of another LSTM (denoted as DeLSTM) for decoding to generate the target sequence $< y_1, y_2, \ldots, y_T >$. In this process, the hidden vectors of the DeLSTM are also conditioned on its input (i.e., $< A_i, B_i, C_i >$ in Figure 1; for a typical language model, this will be $< y_1, y_2, \ldots, y_T >$).

---

[1]We here describe the commonly used Elman-type RNNs [11]; other variants such as Jordan-type RNNs [15] are also available in the community.

3

Consequently, the DeLSTM creates a sequence of hidden states $< h_1^{de}, h_2^{de}, \ldots, h_T^{de} > (h_t^{de} \in R^N)$ for each time step. Next, each of these hidden vectors is fed into a *Softmax* function to produce a distribution over the C possible classes (e.g., words in a vocabulary or dictionary), thus creating a list of word probabilities at each time step t, i.e., $< W_{pro1}^t, W_{pro2}^t, \ldots, W_{proV}^t >$ (V is the size of the dictionary):

$$P(W_{pro}^t = c | EnSen, \vartheta) = \frac{exp(w_c^T h_t^{de})}{\sum_{c=1}^{C} exp(w_c^T h_t^{de})} \qquad (3)$$

where w_c is the weight matrix from the hidden layer to the output layer. These probabilities can be further processed by a $Argmax$ function, resulting in a sequence of output words, namely a sentence $< y_1^i, y_2^i, \ldots, y_T^i >$ (denoted as DeSen$_i$; i indicates the i-th iteration of the DQN, which will discussed in detail later).

The parameter ϑ for the decoder-encoder LSTMs, namely the StateGF function, is tuned to maximize the log probability of a correct decoding sentence Y given the source sentence X, using the following training objective:

$$1/|S| \sum_{(X,Y) \in S} logp(Y|X) \qquad (4)$$

where S is the training set. After training, decoding output sequence can be achieved by finding the most likely output sequence according to the DeLSTM:

$$\hat{Y} = \underset{Y}{\operatorname{argmax}} p(Y|X) \qquad (5)$$

A straight forward and effective method for this decoding search, as suggested by [30], is to deploy a simple left-to-right beam search. That is, the decoder maintains a small number of incomplete sentences. At each timestep, the decoder extends each partial sentence in the beam with every possible word in the vocabulary. As suggested by [30], a beam size of 1 works well.

In this way, feeding the state generate function with EnSen will result in a decoded sentence DeSen. The DQN decoding, which will be discussed next, employs an iteration strategy, so we denote this sentence sequence pair as EnSen$_i$ and DeSen$_i$; here i indicates the i-th iteration of the DQN.

3.2 Iteratively Decoding Sequence with Deep Q-Network

At each decoding iteration i, the DQN considers the sentence pair, namely the EnSen$_i$ and DeSen$_i$ (i.e., $< A, B, C >$ and $< A_i, B_i, C_i >$, respectively, in Figure 1) as its internal state. Also, the ranked words list [2] for each time step t of the DeLSTM is treated as the potential actions by the DQN. From these lists, the DQN learns to predict what actions should be taken in order to accumulate larger long time reward.

In detail, each hidden vector h_t^{de} in the DeLSTM is fed into a neural network (depicted as DQN in figure 1 and graphically illustrated on the top-left subfigure; will be further discussed in Section 3.3). These neural networks learn to approximate the Q-value function given the DQN's current state, which contains the EnSen$_i$ and DeSen$_i$ as well as the word probability list at each time step t of the DeLSTM. The DQN will take the action with the max Q-value in the outputs. Consider, the DQN takes an action, namely selects the t-th time step word $\widehat{y_t^i}$ in iteration i. Then the current state of the DQN will be modified accordingly. That is, the DeSen$_i$ will be modified by replacing the word at time step t, namely replacing y_t^i with $\widehat{y_t^i}$. This process results in a new decoded sentence, namely DeSen$_{i+1}$ (depicted as $< A_{i+1}, B_{i+1}, C_{i+1} >$ in Figure 1). Next, the similarity of the target sentence $< y_1, y_2, \ldots, y_T >$ and the current decoded sentence DeSen$_{i+1}$ is evaluated by a BLEU metric [25], which then assigns a reward r_i to the action of selecting $\widehat{y_t^i}$. Thus, a transition tuple for the DQN contains $[(EnSen_i, DeSen_i), \widehat{y_t^i}, r_i, ([EnSen_i, DeSen_{i+1}])$. In the next iteration of the DQN, the newly generated sentence DeSen$_{i+1}$ is then fed into the DQN to generate the next decoded sentence DeSen$_{i+2}$.

The training of the DQN is to find the optimal weight matrix θ in the neural networks. That is, the Q-network is trained by minimizing a sequence of loss functions $L_i(\theta_i)$ at each iteration i:

$$L_i(\theta_i) = E_{s,a}[(q_i - Q(s, a; \theta_i))^2] \qquad (6)$$

---

[2]This list has the same size as that of the vocabulary; but one can use only a few of the top ranked words.

Algorithm 1 Generating Text with Deep Q-Network

1: Initialize replay memory D; initialize EnLSTM, DeLSTM, and DQN with random weights
2: **Pretraining Encoder-Decoder LSTMs**
3: **for** epoch = 1,M **do**
4:     randomize given training set with sequence pairs $< X, Y >$.
5:     **for** each sequence pair $EnSen^k \in X$ and $TaSen^k \in Y$ **do**
6:         Encode $EnSen^k$ with EnLSTM, and then predict the next token (e.g., word) in $TaSen^k$ with DeLSTM.
7:     **end for**
8: **end for**
9: **Training Q-value function**
10: **for** epoch = 1,U **do**
11:     **for** each sequence pair $EnSen^k \in X$ and $TaSen^k \in Y$ (with length l) **do**
12:         feed $EnSen^k$ into pretrained encoder-decoder LSTMs; obtain the decoded sequence $DeSen_0^k$
13:         **for** iteration i = 1, 2l **do**
14:             **if** random() $< \epsilon$ **then**
15:                 select a random action a_t (e.g., word w) at time step t of $DeSen_i^k$ (selection biases to incorrect decoded tokens)
16:             **else**
17:                 compute $Q(s_i, a)$ for all actions using DQN; select $a_t = argmaxQ(S_i, a)$, resulting in a new token w for the t-th token in $DeSen_i^k$
18:             **end if**
19:             replace $DeSen_i^k$ with w, resulting $DeSen_{i+1}^k$
20:             compute the similarity of $DeSen_{i+1}^k$ and $TaSen^k$, resulting reward score r_i
21:             store transition tuple $[s_i, a_i, r_i, s_{i+1}]$ in replay memory D; $s_i = [EnSen_i^k, DeSen_i^k]$.
22:             random sample of transition $[s_i, a_t, r_i, s_{i+1}]$ in D
23:             **if** $r_i > \sigma$ (preset BLEU score threshold) **then**
24:                 $q_i = r_i$; current sequence decoding successfully complete.
25:             **else**
26:                 $q_i = r_i + \lambda max_{a'}Q(s', a'; \theta_{i-1})$
27:             **end if**
28:             perform gradient descent step on only the DQN network $(q_i - Q(s, a; \theta_i))^2$
29:         **end for**
30:     **end for**
31: **end for**

---

where $q_i = E_{s,a}[r_i + \lambda max_{a'}Q(s', a'; \theta_{i-1})|s, a]$ is the target Q-value, or reward, with parameters θ_{i-1} fixed from the previous iteration. In other words, the DQN is trained to predict its expected future reward. The updates on the parameters $L_i(\theta_i)$ is performed with the following gradient:

$$\nabla_{\theta_i} L_i(\theta_i) = E_{s,a}[2(q_i - Q(s, a; \theta_i))\nabla_{\theta_i}Q(s, a; \theta_i)] \tag{7}$$

After learning the Q-value function, the agent chooses the action with the highest $Q(s, a)$ in order to maximize its expected future rewards when decoding sequences. Quite often, a trade-off between exploration and exploitation strategy is employed for the agent. That is, through following an ϵ-greedy policy, the agent can perform a random action with probability ϵ [31]. Inspired by the recent success of attention mechanisms [3, 13, 20, 28], we here bias the reinforcement learning's attention to explore the sequence portions which are difficult to be decoded. That is, the random actions have more chance to be picked for tokens which were decoded incorrectly from the previous iterations.

3.3 Bidirectional LSTMs for DQN

During decoding, we would like the DQN to have information about the entire input sequence, i.e., $< A_i, B_i, C_i >$ in Figure 1. To attain this goal, we deploy a bidirectional LSTMs [12]. Specifically, for a specific time step t of a given sequence, a Bidirectional LSTM [12] enables the hidden states to summarize time step t's past and future in the sequence. The network deploys two separate hidden layers to precess the data in both directions: one from left to right (forward), and another right to left (backward). At each time step, the hidden state of the Bidirectional LSTM is the concatenation of the forward and backward hidden states, and then fed forwards to the same output layer. That is, Equation 2 for the DQN is implemented as follows (illustrated in the top-left subfigure in Figure 1).

$$\overrightarrow{h_t} = \Gamma(W_{x\overrightarrow{h}}x_t + W_{\overrightarrow{h}\overrightarrow{h}}\overrightarrow{h_{t-1}} + b_{\overrightarrow{h}}) \tag{8}$$

$$\overleftarrow{h_t} = \Gamma(W_{x\overleftarrow{h}}x_t + W_{\overleftarrow{h}\overleftarrow{h}}\overleftarrow{h_{t-1}} + b_{\overleftarrow{h}}) \tag{9}$$

$$h_t = [\overrightarrow{h_t^T}; \overleftarrow{h_t^T}]^T \tag{10}$$

In this scenario, h_t^{de} is equal to $\overrightarrow{h_t}$, namely the forward hidden vectors. The additional information about the input sequence $< A_i, B_i, C_i >$ is further summarized by the backward hidden vectors $\overleftarrow{h_t}$.

3.4 BLEU Score for DQN Reward

Reward is calculated based on the closeness between the target sentence $< y_1, y_2, \ldots, y_T >$ and the decoded output sentence (i.e., $DeSen$) after the DQN takes an action. We compute the similarity of this sentence pair using the popular score metric in statistical translation. Specifically, we obtain a BLEU [25] score between these two sentences. We measure the score difference between the current iteration and the previous iteration. If the difference is positive, then a reward of +1 is assigned; if negative then -1 as reward; otherwise, it is zero. Note that, since we here conduct a sentence level comparison, we adopt the smoothed version of BLEU [17]. Unlike the BLEU, the smoothedBLEU avoids giving zero score even when there are not any 4-gram matches in the sentence pair.

3.5 Empirical Observations on Model Design

Separating State Generation Function from DQN Our experiments suggest that separating the state generation function from the DQN networks is beneficial. The aim here is to have a deterministic network for generating states from a sequence pair. That is, for any given input pair to the encoder-decoder LSTMs network, namely the state generation function StateGF, we will always have the same decoded output sequence. Our empirical studies indicate that this is a very important for successfully training the DQN for decoding text. Our intuitive explanation is as follows.

Using DQN to approximate the Q-value function, intuitively, equals to train a network against moving targets because here the network's targets depend on the network itself. Suppose, for a given input feed, the StateGF would generate a different output sequence each time for the DQN. In this scenario, the DQN network has to also deal with a moving state function involving text with *very high dimensionality*. Intuitively, here the DQN agent is living in a changing environment Ξ. As a result, it may be very difficult for the DQN to learn to predict the Q-value, since, now, all the states and rewards are unstable, and change even for the same input feed.

Pre-training the State Generation Function Two empirical techniques are employed to ensure that we have a deterministic network for generating states for DQN. Firstly, we deploy a pre-training technique. Specifically, we pre-train the state generation function StateGF with the input sequence X as the EnLSTM's input and target sequence Y as the DeLSTM's input. After the training converges, the networks' weights will be fixed when the training of the DQN network starts. Secondly, during training the DQN, the input sequence is fed into the EnLSTM, but the decoded sequence from the previous iteration is used by the DeLSTM as input (indicated as dot line in Figure 1). In this stage, only the red portions of Figure 1 are updated. That is, the reward errors from the DQN networks are not backpropagated to the state generation functions.

Updating with Replay Memory Sampling Our studies also indicate that, performing updates to the Q-value function using transitions from the current training sentence causes the network to strongly overfit the current input sentence. As a result, when a new sentence is fed in for training, it may always predict the previous sentence used for training. To avoid this correlation issue, a replay memory strategy is applied when updating the DQN. That is, the DQN is updated by transition tuples which may be different from the current input sequence.

To this end, for each action the DQN takes, we save its transition tuple in the replay memory pool, including the EnSen$_i$, DeSen$_i$, DeSen$_{i+1}$, r_i, and a_i. When updating the DQN, we then randomly sample a transition tuple from the replay memory pool. More sophisticated replay memory update could be applied here; we would like to leave it for future work. For example, one can use the priority sampling of replay technique [22]. That is, transitions with large rewards have more chance to be chose. In our case, we can bias the selection to transitions with high BLEU scores.

Importance of Supervised Softmax Signal We also conduct experiments without the supervised $Softmax$ error for the network. That is, the whole network, including the LSTMs and DQN, only receive the error signals from the Q-value predictions. We observed that, without the supervised signal the DQN was very difficult to learn. The intuition is as follows. Firstly, as discussed before, for decoding text, which typically involves a very large number of potential states and actions, it is very challenge for the DQN to learn the optimal policy from both a moving state generation function and a moving Q-value target function. Secondly, the potential actions for the DQN, namely the word probability list for each output of the DeLSTM is changing and unreliable, which will further complicate the learning of the DQN.

Simultaneously Updating with Both Softmax and Q-value Error If during training the DQN, we not only update the DQN as discussed previously, but also update the state generation functions, i.e., the encoder-decoder LSTMs. We found that the network could be easily bias to the state generation functions since the $Softmax$ error signal is very strong and more reliable (compared to the moving target Q-value function), thus the DQN may not be sufficiently tuned. Of course, we could bias towards the learning of DQN, but this would introduce one more tricky parameter for tuning. In addition, doing so, we have an indeterministic state generation function again.

4 Experiments

4.1 Task and Dataset

Our experimental task here is to train a network to regenerate natural sentences. That is, given a sentence as input, the network first compresses it into a fixed vector, and then this vector is used to decode the input sentence. In other words, the X and Y in Algorithm 1 are the same. In our experiment, we randomly select 12000 sentences, with max length of 30, from the Billion Word Corpus [7]. We train our model with 10000 sentences, and then select the best model with the validation data which consist of 1000 sentences. We then test our model with 1000 seen sentences and 1000 unseen sentences. The seen test set is randomly sampled from the training set.

4.2 Training and Testing Detail

For computational reason, we used an one-layer LSTM for the encoder-decoder LSTMs as well as the backward LSTM in the DQN, both with 100 memory cells and 100 dimensional word embeddings. We used a Softmax over 10000 words (which is the size of the vocabulary we used in the experiments) at each output (i.e., time step t) of the DeLSTM. We initialized all of the LSTMs parameters with the uniform distribution between -0.15 and +0.15, including the word vectors. We used Adaptive Stochastic Gradient Descent (AdaSGD) [9] without momentum, with a starting learning rate of 0.05. Although LSTMs tend to not suffer from the vanishing gradient problem, they can have exploding gradients. Thus we employ the gradient norm clip technique [26] with a threshold of 15. We used both L2 regularization (with a weight decay value of 0.00016) and dropout (with a rate of 0.2) to avoid overfitting the networks.

We first pretrain the encoder-decoder LSTMs with both the target sentence as input. After the training converges, we then start to train the DQN. When training the DQN, we turn off the drop out in the encoder-decoder LSTMs, so that we have a deterministic network to generate states and the lists of word probabilities for the DQN. In addition, we scale down the epsilon ϵ to 0.1 after 2000000 iterations. In other words, most of actions at the beginning of the DQN training were random, and then became more greedy towards the end of the training. For each sentence with length of l, we allow DQN to edit the sentence with $2l$ iterations, namely taking $2l$ actions for the decoding. The sentence decoded in each iteration will be saved in a replay memory with a capacity of 500000. The discount factor λ was set to 0.95. Also, the BLEU score threshold σ for indicating decoding success was set to 0.92. For the initial states of the bi-directional LSTMs in the DQN, we used the fixed vector generated by the LSTM encoder.

In testing phase, we also run the DQN for each sentence with $2l$ steps. Also, in our experiment, we used only the word with the max probability on each of the T lists as the potential actions for the DQN. Since the maximal length of a sentence in our experiment is 30, the DQN has at most 31 output nodes. Namely, the DQN can choose one of the 30 top words, each corresponding to a time step at the DeLSTM, as its action, or take the 31st action which indicates not modification is needed.

We compared our strategy with an encoder-decoder LSTM network used in [30] for machine translation. This baseline decoder searches for the most likely output sequence using a simple left-to-right beam search technique. As suggested by [30], a beam size of 1 worked well. We adopt this approach as our decoding baseline. All our experiments were run on a NVIDIA GTX TitanX GPU with 12GB memory. We report the average SmoothedBLEU score for all the testing sentences.

4.3 Experimental Results

The evolutions of the training for the state generation function StateGF and DQN are depicted in Figure 2, and the main testing results are presented in Table 1.

From the training curves for both the encoder-decoder LSTMs and DQN as depicted in Figure 2, we can see that both of the trainings converged very well. For the LSTMs training, the average cost
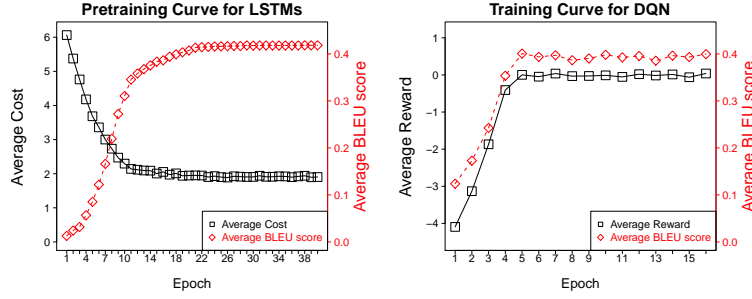
Figure 2: The evolution of cost for training the StateGF and reward for training the DQN.

was steadily decreasing, and the average BLEU score was gradually increasing. Both the two curves then stabilized after 20 iterations. Also, the right subfigure in Figure 2 indicates that, the reward obtained by the DQN was negative at the beginning of the training and then gradually moved to the positive reword zone. The negative rewards are expected here because most of the DQN's actions were random at the beginning of the training. Gradually, the DQN knew how to decode the sentences to receive positive rewards. As shown in the right figure, the training of the DQN converged in about 6 epochs. These results indicate that both the state generation functions and the DQN were easy to be trained.

Testing systems	LSTM decoder	DQN
Average SmoothedBLEU on sentences IN the training set	0.425	0.494
Average SmoothedBLEU on sentences NOT in the training set	0.107	0.228

Table 1: Experimental results for decoding the seen and unseen sentences in testing.

In Table 1, we show the testing results, in terms of average SmoothedBLEU obtained, for both the seen 1000 and unseen 1000 sentences. We can observe that, although the results achieved by the DQN on the seen data were only slightly better than that of the baseline LSTMs network, for the unseen data the DQN meaningfully outperformed the baseline. Our further analysis suggests the follows. With the seen data, the DQN decoder tended to agree with the LSTM decoder. That is, most of the time, its decision was "no modification". As for the unseen data, because the DQN's exploration strategy allows it to learn from many more noisy data than the LSTMs networks did, so the DQN decoder was able to tolerate better to noise and generalize well to unseen data. Intuitively, the application of the DQN here also has the effect of generating synthetic sequential text for the training of the DQN decoder, due to its exploration component.
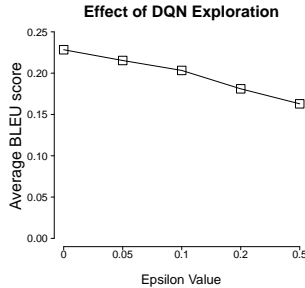

Figure 3: Impact, in terms of BLEU score obtained, of the DQN's exploration in the testing phase.

We also conducted experiments to observe the behaviors of the DQN for exploration; here we only considered the unseen testing data set. That is, we enabled the DQN to follow an ϵ-greedy policy with $\epsilon = 0, 0.05, 0.1, 0.2, 0.5$, respectively. In other words, we allowed the agent to choose the best actions according to its Q-value function 100%, 95%, 90%, 80%, and 50% of the time. The experimental results, in terms of BLEU score obtained, are presented in Figure 3. From Figure 3,

we can conclude that the exploration strategy in testing time did not help the DQN. The results here indicate that allowing the DQN to explore in testing time decreased its predictive performance, in terms of BLEU score obtained

5   Related Work

Recently, the Deep Q-Network (DQN) has been shown to be able to successfully play Atari games [14, 21, 24]. Trained with a variant of Q-learning [34], the DQN learns control strategies using deep neural networks. The main idea is to use deep learning to automatically generate informative features to represent the internal states of the environment where the software agent lives, and subsequently approximate a non-linear control police function for the learning agent to take actions. In addition to playing video games, employing reinforcement learning to learn control policies from text has also be investigated. Applications include interpreting user manuals [6], navigating directions [2, 16, 18, 33] and playing text-based games [5, 10, 23]. Also, DQN has recently been employed to learn memory access patterns and rearrange a set of given words [35].

Unlike the above works, our research here aims to decode natural text with DQN. In addition, we employ an encoder-decoder LSTM network to not only generalize informative features from text to represent the states of DQN, but also create a list of potential actions from the text for the DQN.

6   Conclusion and Future Work

We deploy a Deep Q-Network (DQN) to embrace an iterative decoding strategy for sequence to sequence learning. To this end, an encoder-decoder LSTM network is employed to automatically approximate internal states and formulate potential actions for the DQN. In addition, we incorporate an attention mechanism into the reinforcement learning's exploration strategy. Such exploration, intuitively, enables the decoding network to learn from many synthetic sequential text generated during the decoding stage. We evaluate the proposed method with a sentence regeneration task. Our experiments demonstrate our approach's promising performance especially when decoding unseen sentences, in terms of BLEU score obtained. This paper also presents several empirical observations, in terms of model design, in order for successfully decoding sequential text with DQN.

In the future, allowing the DQN to pick from the top n words from the list at each time step t of the DeLSTM would be further studied. Furthermore, we would like to experiment with sophisticated priority sampling techniques for the DQN training. In particular, we are interested in applying this approach to statistical machine translation.

References

[1] C. Amato and G. Shani. High-level reinforcement learning in strategy games. In *AAMAS '10*, pages 75–82, 2010.

[2] Y. Artzi and L. Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. *TACL*, 1:49–62, 2013.

[3] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[4] S. R. K. Branavan, D. Silver, and R. Barzilay. Learning to win by reading manuals in a monte-carlo framework. In *ACL '11*, pages 268–277, 2011.

[5] S. R. K. Branavan, D. Silver, and R. Barzilay. Learning to win by reading manuals in a monte-carlo framework. *CoRR*, abs/1401.5390, 2014.

[6] S. R. K. Branavan, L. S. Zettlemoyer, and R. Barzilay. Reading between the lines: Learning to map high-level instructions to commands. In *ACL '10*, pages 1268–1277, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[7] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson. One billion word benchmark for measuring progress in statistical language modeling. In *INTERSPEECH 2014*, pages 2635–2639, 2014.

[8] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[9] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[10] J. Eisenstein, J. Clarke, D. Goldwasser, and D. Roth. Reading to learn: Constructing features from semantic abstracts. In *EMNLP '09*, pages 958–967, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[11] J. L. Elman. Finding structure in time. *COGNITIVE SCIENCE*, 14(2):179–211, 1990.

[12] A. Graves, A. Mohamed, and G. E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013.

[13] K. Gregor, I. Danihelka, A. Graves, and D. Wierstra. DRAW: A recurrent neural network for image generation. *CoRR*, abs/1502.04623, 2015.

[14] M. J. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015.

[15] M. I. Jordan. Chapter 25 - serial order: A parallel distributed processing approach. In J. W. Donahoe and V. P. Dorsel, editors, *Neural-Network Models of CognitionBiobehavioral Foundations*, volume 121 of *Advances in Psychology*, pages 471 – 495. North-Holland, 1997.

[16] T. Kollar, S. Tellex, D. Roy, and N. Roy. Toward understanding natural language directions. In *Proceedings of the 5th ACM/IEEE International Conference on Human-robot Interaction*, HRI '10, pages 259–266, Piscataway, NJ, USA, 2010. IEEE Press.

[17] C.-Y. Lin and F. J. Och. Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In *ACL '04*, 2004.

[18] C. Matuszek, E. Herbst, L. Zettlemoyer, and D. Fox. Learning to parse natural language commands to a robot control system. In *Experimental Robotics*, pages 403–415. Springer International Publishing, 2013.

[19] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *INTERSPEECH 2010*, pages 1045–1048, 2010.

[20] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu. Recurrent models of visual attention. *CoRR*, abs/1406.6247, 2014.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.

[22] A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. In *Machine Learning*, pages 103–130, 1993.

[23] K. Narasimhan, T. Kulkarni, and R. Barzilay. Language understanding for text-based games using deep reinforcement learning. *CoRR*, abs/1506.08941, 2015.

[24] J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. P. Singh. Action-conditional video prediction using deep networks in atari games. *CoRR*, abs/1507.08750, 2015.

[25] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: A method for automatic evaluation of machine translation. In *ACL '02*, pages 311–318, 2002.

[26] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. 2013.

[27] D. Silver, R. Sutton, and M. Müller. Reinforcement learning of local shape in the game of go. In *IJCAI'07*, pages 1053–1058, San Francisco, CA, USA, 2007.

[28] S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus. Weakly supervised memory networks. *CoRR*, abs/1503.08895, 2015.

[29] M. Sundermeyer, R. Schlüter, and H. Ney. Lstm neural networks for language modeling. In *INTERSPEECH*, pages 194–197, 2012.

[30] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[31] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[32] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *CoRR*, abs/1411.4555, 2014.

[33] A. Vogel and D. Jurafsky. Learning to follow navigational directions. In *ACL '10*, pages 806–814, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[34] C. J. C. H. Watkins and P. Dayan. Q-learning. In *Machine Learning*, pages 279–292, 1992.

[35] W. Zaremba and I. Sutskever. Reinforcement learning neural turing machines. *CoRR*, abs/1505.00521, 2015.