A Natural Language Model for Generating PDDL

Nisha Simon, Christian Muise
20nis@queensu.ca, christian.muise@queensu.ca
Queen's University

Abstract

Language generation in various domains has drawn a large amount of interest in recent years. This paper studies language generation in the context of generating planning specifications in the syntax typically used for this task: the Planning Domain Definition Language (PDDL). The goal of this preliminary work is to predict the next completion in PDDL code, based on previous and surrounding text. Generating valid PDDL code is a key component in creating robust planners. Thus, the ability to generate PDDL code will be extremely useful to PDDL practitioners for the purpose of solving planning problems. It further opens the door to providing a source of inspiration for the modeller. The main contribution of our approach is a language model built using Recurrent Neural Networks (RNNs) that is trained on existing PDDL domains, which can be used to generate PDDL-like code. We train our model on a corpus of publicly available PDDL files from api.planning.domains, and evaluate our approach in the setting of PDDL auto-prediction for some of the more common domains. We found that code-like generation is possible, although fluency can be improved.

1 Introduction

Planning problems include a start state, an end (goal) state and a set of allowable actions. Solving a planning problem involves selecting the sequence of actions that are required in order to move from the start state to the goal state. Planning Domain Definition Language (PDDL) is used to define planning problems. We focus on the task of PDDL Language generation, which means that given some preliminary text, we predict the next completion using PDDL syntax. We create text (PDDL code) that flows coherently from previously seen text in the domain of the planning model. It is to be noted that the goal is not the creation of a well defined plan, but rather that of the planning problem specification. The main contribution of this paper is a language model capable of generating PDDL text given the context. Moving towards auto-completion serves to ease the strain on the planning designer since being able to generate PDDL code removes the burden of coding from the planning designer, who can now focus on creating the actual content of the plan and not on the creation of the relevant PDDL code.

The data source that was used for training the language model is api.planning.domains (Muise 2016). The models that were used for training were '*bartender*', which simulates a bartender taking and serving drink orders, '*ferry*' which simulates a passenger ferry, and '*movie*' which simulates a movie watching experience complete with snacks. A network of Recurrent Neural Networks (RNNs) was used to build the model. The software for the project was based on the open source deep learning library, *PyTorch*.

The layout of the paper is as follows: We briefly describe the background of our research area in section 2. The system architecture and implementation details are described in section 3. Section 5 places the project in the context of other research in this area. Finally, our contributions and evaluation of the results are then laid out in section 4, followed by the conclusion and ideas for future work in section 6.

2 Background

Recurrent Neural Networks (RNNs) have been used in deep learning applications for several language tasks, and have proven successful for generating variable length sequences. (Pascanu et al. 2014). RNNs are neural networks that consist of multiple levels of nodes which form input layers, output layers, and hidden layers. If the inputs are represented as x_t then the outputs can be shown as y_t and hidden states are h_t where *t* represents time (Pascanu et al. 2014). The layers are trained on given inputs in training mode, and then they are run in testing mode in order to produce the desired outputs.

Given a certain starting symbol or character, the RNN can be used to predict the next most likely character. In other words, we model the distribution over a sequence (Pascanu et al. 2014) based on the highest probable output value and on the hidden states as shown in algorithm 1. We obtain the next letter based on the highest probable output value and the next hidden state, and then add it to the output. This process is repeated until either an end of sentence (EOS) token is obtained or until the maximum allowed line length (that can be set to a particular suitable value) has been reached.

3 Approach

The premise of the project is based on auto-regressive language generation, which is to say that we assume that the probability of a word or character depends on the conditional probability of the previous words or characters. Our input that was used in this project comes from the PDDL models at api.planning.domains (Muise 2016). We use an RNN

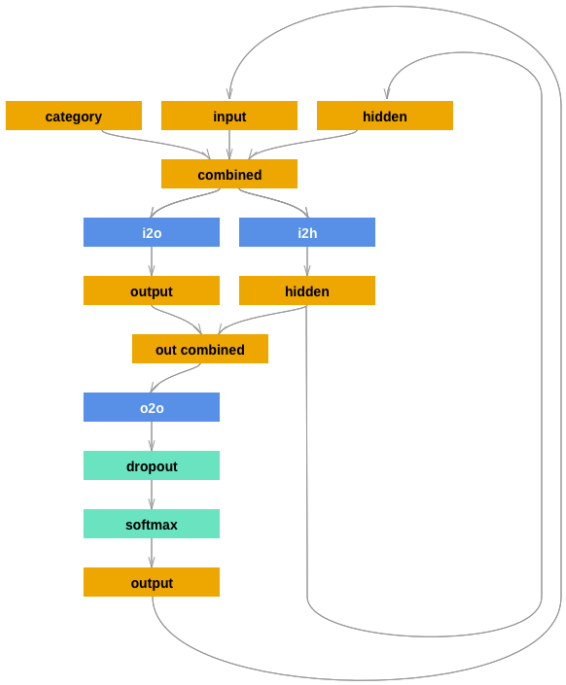

Figure 1: Network Structure[1]

character-level language model to generate PDDL-code-like output lines. The various stages of the model are data extraction, data cleaning (removing unwanted characters, such as punctuation, as required), data preprocessing for RNNs, training, optimization and prediction. Once the training is completed the predictor can then be run multiple times using different starting characters for the output string.

The particular type of PDDL problem domain acted as the '*category*'. for example '*ferry*', '*bartender*', or '*movie*'. The input lines data was read in from the appropriate PDDL code (which had been first transformed into plain text files) and then a dictionary of category-lines, which is a list of lines per category, was built. The RNN model was then created with neural network layers as depicted in Figure 1 and which was adapted from (Robertson 2017).[1] The category and input lines were then converted into one-hot vectors. The output was calculated as the probability of the next letter. For those familiar with general deep learning configurations, a drop-out layer randomly set part of the input to 0, in order to prevent overfitting. The system was trained over a large number of iterations and the Adam optimizer was used to adjust parameters as needed. The objective was to minimize the loss value, which was calculated as the difference between the actual 'correct' output and the generated output.

To perform the prediction, we gave the network a starting letter and then asked it to predict what the next letter was likely to be. We repeated this process by feeding in that prediction as the next letter, and continued until the End of Sentence (EOS) token was reached. This was accomplished as shown in algorithm 1.

[1]Figure reproduced from (Robertson 2017)

Algorithm 1: Building an output string

Input: The selected starting letter
Result: Return the final output line
Create an output string that begins with the given starting letter;
while *current line length* $<$ *the maximum allowed output length* **do**
  Feed the current letter to the network;
  if *next letter is not EOS,* **then**
    Get the next most likely letter ;
    Add this next letter to the output and then continue;
  else
    stop;
  end
end

Listing 1 shows the partial process of how an output line is generated character by character according to algorithm 1 as described above. From the selected starting symbol (or letter) of '*(*' which in this case is a left parenthesis, the output line is built up character by character to form the final line '*(cheese z14)*'. The partial snippet shown here demonstrates that the system has so far predicted the output line of '*(chee*' and it then predicts that the next most probable letter is likely to be '*s*', followed by '*e*'. The process continues until the final output line of '*(cheese z14)*' which is found to be a valid PDDL-code line, is generated.

```
...
new output is (chee

The next top 5 target tensors in decreasing
    order are tensor([[-5.3006e-03, -7.0581e
    +00, -7.0581e+00, -7.0581e+00, -7.0581e
    +00]])
The selected (most probable) letter is s
new output is (chees

The next top 5 target tensors in decreasing
    order are tensor([[-5.8618e-03, -7.0867e
    +00, -7.0867e+00, -7.0867e+00, -7.0867e
    +00]])
The selected (most probable) letter is e
new output is (cheese
...
#Final Output line
(cheese z14)
```

Listing 1: Building an output line

The system parameters and hyper-parameters are shown in Table 1. Initially the SGD optimizer was used. However, the Adam optimizer provided superior results over the SGD optimizer. The number of iterations that were used was slowly increased, starting from a smaller value and then increasing to larger values, in order to improve the accuracy of the produced output lines. 100,000 iterations were sufficient to produce PDDL-code-like lines for this dataset. The Learning Rate was set at 0.0005, as higher values proved to

be unsuitable for producing accurate output, and lower values resulted in a loss curve that did not decreases as rapidly as desired. The layers of the Neural Network, which used a hidden size of 128, can be seen in graphical form in Figure 1. The loss function used was NLLLoss since behind the scenes, this was partially a classification task that was based on the category that was provided.

Name	Value or Type
Optimizer	Adam
Iterations	100,000
Learning Rate	0.0005
Hidden Size	128
Batch Size	1
Layers	2 + 2
Loss	NLLLoss
Maximum output line length	70

Table 1: System Parameters and Hyperparameters

4 Evaluation

The system successfully produces PDDL-like statements according to the domain on which it was trained. For example: *(contains shot2)*, where it can be noted that the system has learned the fact that brackets should be matched, and also that items can be listed as numeric lists such as *shot2* etc. The system was trained on a large number of iterations, and as expected, the loss function decreased as the number of iterations increased and the system learnt the required output, which is shown in Figure 2.

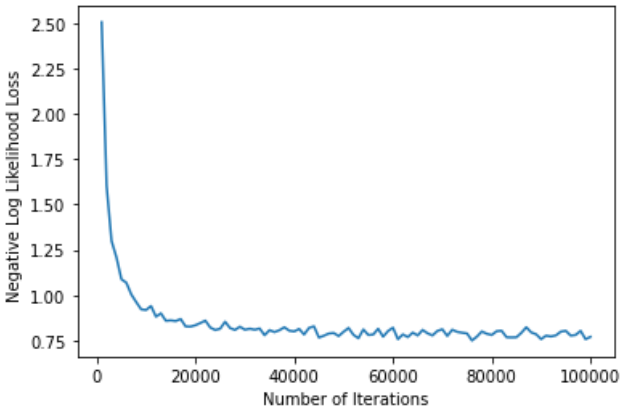

Figure 2: Loss Function

As can be seen from Figure 2, increasing the number of iterations improves the loss value. We used log likelihood loss as our loss function. The smaller the loss value, the smaller the difference between the correct output and the generated output, and the greater the accuracy in the material that the system has been able to learn. Due to the the large size of the model and limited GPU memory, a batch size of one was used in our experiments. It is to be expected that the availability of a larger dataset would result in more accurate predictions and output. Nevertheless it can be seen that the system learns successfully, being able to identify key terms from each domain such as *'cocktail', 'shot', 'dispenser', 'chips', 'pop', 'cheese', 'crackers', 'dip', 'car'* and '*level*'. In addition, as can be seen from Table 2 and Table 3, the system can generate suitable output lines based on the given input. Table 2 shows actual lines of PDDL code and Table 3 shows that the system is able to generate not just the appropriate required keywords, but partial syntax as well, although the level of fluency could be improved in order to bring the generated text even closer to replicating actual PDDL code.

```
shaker1 - shaker
left right - hand
shot1 shot2 shot3 shot4 - shot
ingredient1 ingredient2 ingredient3 - ingredient
cocktail1 cocktail2 cocktail3 - cocktail
dispenser1 dispenser2 dispenser3 - dispenser
(dip d3)
(dip d2)
(dip d1)
(pop p5)
(pop p4)
(pop p3)
(pop p2)
(pop p1)
(cheese z5)
(cheese z4)
(location l8)
(location l9)
(car c0)
(car c1)
(car c2)
(car c3)
(car c4)
```

Table 2: Sample PDDL Input Code lines

5 Related Work

RNNs are networks that take an input x_t and produce an output y_t via hidden states h_t where the subscript t represents time (Pascanu et al. 2014). The use of RNNS to create deep networks is described in (Pascanu et al. 2014). The authors demonstrate that deep RNNs are considered to be more effective in language modelling, as opposed to conventional shallow RNNs.

In their paper '*A Survey of the Usages of Deep Learning for Natural Language Processing*' the authors describe the use of deep learning methods for language generation (Otter, Medina, and Kalita July 2019). Only small amounts of input data are needed in order to serve as a guide for the system to produce poetry, for example. They affirm that a challenge to be overcome in computer generated language is that while the output may be difficult to distinguish from human-generated text, it is still likely to be of lower quality

Category	Line
Bartender PDDL	shot1 cocktail1 ingredient2) cocktail-part2 cocktail1 ingredient2) contabns shot1) shaker1) (clean s9 k1) cocktail-part1 cocktail1 ingredient3) shot1 shot2 shot3 shot2 shot2 shot2 shot2 shot2 shot2 shot2 shot2 shot2 dispenses dispenser1 ingredient1) (contains6) shaker1 - shaker (contains shot2)
Movie PDDL	chips c11) chips c11) pop p14) (chips c11)) (cheese z1) (crackers k10) (dip d17) (cheese z5)
Ferry PDDL	(not-eq l10 l1) cl0 l1 l2 - level :init (car c6) (not-eq l6 l11)

Table 3: Generated Output Lines

than that produced by human authors. A lack of coherence and creativity may also be an issue. Some possible methods for text generation that are suggested in the paper include a tiered network of Convolutional Neural Networks (CNNs), Generative Adversarial Networks (GANs), and variational autoencoders (VAEs).

Katz et al. describe the need for automating the work of the human expert in machine learning (2020), especially in the area of optimizing hyper-parameters and selecting the best configurations. The aspect of learning that is studied in their paper is Context Free Grammars (CFGs), since grammars are keys to language generation, and sentence generation can be viewed as a planning problem, as described in (Koller and Hoffmann 2010). The goal of (Katz et al. 2020) was to produce plans that better reflect user preferences.

In (Young et al. 2018) the authors describe the use of Recurrent Neural Networks (RNNs) for text generation. The strength of RNNs lies in their ability to process sequences of text of arbitrary length. More complex variations of RNNS, such as Long Short-Term Memory (LSTM) (Greff et al. 2016) and Gated Recurrent Units (GRUs) (Bansal, Belanger, and McCallum 2016), are even more powerful. LSTM encoder-decoder models can be used for language generation by mapping one sequence onto another. Attention based models such as the Transformer, which consists of stacked layers of encoder and decoder components, further improve the system's performance (Vaswani et al. 2017).

The use of 'narrative generation as a novel application of planning techniques' is studied in (Polceanu et al. 2020). A sequence to sequence encoder-decoder architecture is used. The authors stress that 'model performance on the task of generating novel narrative plots relies not only on the percentage of well-formed plans, but on a balance between the accuracy, the capability to generate a reasonable number of novel sequences with desirable narrative properties and the ability to capture longterm dependencies.' (Polceanu et al. 2020). Well-formedness was imposed as a constraint in their paper, and it should be noted that this is less of a concern for story-telling, where artistic licence may be taken, than it is for the current project, which is code generation and which therefore relies more heavily on syntactic accuracy.

Andrej Karpathy provides a summary of the uses of RNN for text generation in his highly cited online article, "*The Unreasonable Effectiveness of Recurrent Neural Networks*" (Karpathy 2015). Data can be processed sequentially, with the current outputs being dependent not just on the current inputs, but on the previous inputs as well. The RNNs' hidden state is updated at each step. The RNN is given a chunk of text and then it is asked to predict the next character that is likely to appear in the sequence. Karpathy demonstrates that this technique can be used to create a wide variety of text, including a facsimile of dialogue from a Shakespearean play as well as program-like code.

6 Conclusion

We have demonstrated a preliminary step in language generation in the context of generating planning specifications in the PDDL syntax. The goal of this initial work, which was to predict the next completion in PDDL code based on previous text, was achieved through the novel use of multi-layered RNNS in the PDDL generation setting. Since generating valid PDDL is a key step in the life-cycle of automated planning research, this ability to generate PDDL code may provide valuable assistance to planning practitioners. After training our model on a corpus of publicly available PDDL files, we evaluated our approach in the setting of PDDL auto-prediction for some of the more common domains.

The current project focused on three main domains, '*bartender*', '*ferry*' and '*movie*'. We found that code-like generation is possible, although fluency can be improved. It is our hope that the results of this project will be useful for knowledge engineering systems and will serve as the foundation for intelligent auto-completion on the freely available PDDL editors used by the planning community.

Future work will include the addition of a greater number of domains and also the use of larger datasets in order to improve the fluency of the generated output. A further extension of the system will be to provide suggestions for the next step as an overall model is being generated. At a later stage, the goal will be to generate new models of the same class as the training model, but using different objects, for example the '*bartender*' model could be extended to '*food server*'. A visualization component which can be used to demonstrate the probability of the next item being calculated will also be added to the system to provide improved functionality.

References

Bansal, T.; Belanger, D.; and McCallum, A. 2016. Ask the gru: Multi-task learning for deep text recommendations. In *proceedings of the 10th ACM Conference on Recommender Systems*, 107–114.

Greff, K.; Srivastava, R. K.; Koutník, J.; Steunebrink, B. R.; and Schmidhuber, J. 2016. LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems* 28(10): 2222–2232.

Karpathy, A. 2015. The Unreasonable Effectiveness of Recurrent Neural Networks. URL https://karpathy.github.io/2015/05/21/rnn-effectiveness/.

Katz, M.; Ram, P.; Sohrabi, S.; and Udrea, O. 2020. Exploring Context-Free Languages via Planning: The Case for Automating Machine Learning. *Proceedings of the International Conference on Automated Planning and Scheduling* 30(1): 403–411.

Koller, A.; and Hoffmann, J. 2010. Waking up a sleeping rabbit: On natural-language sentence generation with FF. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

Muise, C. 2016. Planning.Domains. In *The 26th International Conference on Automated Planning and Scheduling - Demonstrations*.

Otter, D. W.; Medina, J. R.; and Kalita, J. K. July 2019. A Survey of the Usages of Deep Learning for Natural Language Processing. *IEEE Transactions On Neural Networks And Learning Systems* 20(10).

Pascanu, R.; Gülçehre, Ç.; Cho, K.; and Bengio, Y. 2014. How to Construct Deep Recurrent Neural Networks. In Bengio, Y.; and LeCun, Y., eds., *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*.

Polceanu, M.; Porteous, J.; Lindsay, A.; and Cavazza, M. 2020. Narrative Plan Generation with Self-Supervised Learning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence*.

Robertson, S. 2017. NLP From Scratch: Generating Names with a Character-Level RNN. URL https://pytorch.org/tutorials/intermediate/char_rnn_generation_tutorial.html.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 6000–6010.

Young, T.; Hazarika, D.; Poria, S.; and Cambria, E. 2018. Recent trends in deep learning based natural language processing. *IEEE Computational Intelligence magazine* 13(3): 55–75.