Latent Predictor Networks for Code Generation

Wang Ling♢ Edward Grefenstette♢ Karl Moritz Hermann♢
Tomas Kocisky♢♣ Andrew Senior♢ Fumin Wang♢ Phil Blunsom♢♣

♢Google DeepMind ♣University of Oxford
{lingwang,etg,kmh,tkocisky,andrewsenior,awaw,pblunsom}@google.com

Abstract

Many language generation tasks require the production of text conditioned on both structured and unstructured inputs. We present a novel neural network architecture which generates an output sequence conditioned on an arbitrary number of input functions. Crucially, our approach allows both the choice of conditioning context and the granularity of generation, for example characters or tokens, to be marginalised, thus permitting scalable and effective training. Using this framework, we address the problem of generating programming code from a mixed natural language and structured specification. We create two new data sets for this paradigm derived from the collectible trading card games Magic the Gathering and Hearthstone. On these, and a third preexisting corpus, we demonstrate that marginalising multiple predictors allows our model to outperform strong benchmarks.

1 Introduction

The generation of both natural and formal languages often requires models conditioned on diverse predictors [1, 2]. Most models take the restrictive approach of employing a single predictor, such as a word softmax, to predict all tokens of the output sequence. To illustrate its limitation, suppose we wish to generate the answer to the question "Who wrote The Foundation?" as "The Foundation was written by Issac Asimov". The generation of the words "Issac Asimov" and "The Foundation" from a word softmax trained on annotated data is unlikely to succeed as these words are sparse. A robust model might, for example, employ one predictor to copy "The Foundation" from the input, and a another one to find the answer "Issac Asimov" by searching through a database. However, training multiple predictors is in itself a challenging task, as no annotation exists regarding the predictor used to generate each output token. Furthermore, predictors generate segments of different granularity, as database queries can generate multiple tokens while a word softmax generates a single token. In this work we introduce *Latent Predictor Networks (LPNs)*, a novel neural architecture that fulfills these desiderata: at the core of the architecture is the exact computation of the marginal likelihood over latent predictors and generated segments allowing for scalable training.

We introduce a new corpus for the automatic generation of code for cards in Trading Card Games (TCGs), on which we validate our model. TCGs, such as Magic the Gathering (MTG) and Hearthstone (HS), are games played between two players that build decks from an ever expanding pool of cards. Examples of such cards are shown in Figure 1. Each card is identified by its attributes (e.g., name and cost) and has an effect that is described in a text box. Digital implementations of these games implement the game logic, which includes the card effects. This is attractive from a data extraction perspective as not only are the data annotations naturally generated, but we can also view the card as a specification communicated from a designer to a software engineer.

This dataset presents additional challenges to prior work in code generation [2, 3, 4, 5, 6], including the handling of structured input—i.e. cards are composed by multiple sequences (e.g., name and

Figure 1: Example MTG and HS cards.

description)—and attributes (e.g., attack and cost), and the longevity of the generated sequences. Thus, we propose an extension to attention-based neural models [7] to attend over structured inputs. Finally, we propose a code compression method to reduce the size of the code without impacting the quality of the predictions.

Experiments performed on our new datasets, and a further pre-existing one, suggest that our extensions outperform strong benchmarks.

The paper is structured as follows: We first describe the data collection process (Section 2) and formally define our problem and our baseline method (Section 3). Then, we propose our extensions, namely, the structured attention mechanism (Section 4) and the LPN architecture (Section 5). We follow with the description of our code compression algorithm (Section 6). Our model is validated by comparing with multiple benchmarks (Section 7). Finally, we contextualize our findings with related work (Section 8) and present the conclusions of this work (Section 9).

2 Dataset Extraction

We obtain data from open source implementations of two different TCGs, MTG in Java[1] and HS in Python.[2] The statistics of the corpora are illustrated in Table 1. In both corpora, each card is implemented in a separate class file, which we strip of imports and comments. We categorize the content of each card into two different groups: *singular fields* that contain only one value; and *text fields*, which contain multiple words representing different units of meaning. In MTG, there are six singular fields (attack, defense, rarity, set, id, and health) and four text fields (cost, type, name, and description), whereas HS cards have eight singular fields (attack, health, cost and durability, rarity, type, race and class) and two text fields (name and description). Text fields are tokenized by splitting on whitespace and punctuation, with exceptions accounting for domain specific artifacts (e.g., Green mana is described as "{G}" in MTG). Empty fields are replaced with a "NIL" token.

The code for the HS card in Figure 1 is shown in Figure 2. The effect of "drawing cards until the player has as many cards as the opponent" is implemented by computing the difference between the

[1] github.com/magefree/mage/
[2] github.com/danielyule/hearthbreaker/

	MTG	HS
Programming Language	Java	Python
Cards	13,297	665
Cards (Train)	11,969	533
Cards (Validation)	664	66
Cards (Test)	664	66
Singular Fields	6	4
Text Fields	8	2
Words In Description (Average)	21	7
Characters In Code (Average)	1,080	352

Table 1: Statistics of the two TCG datasets.

players' hands and invoking the draw method that number of times. This illustrates that the mapping between the description and the code is non-linear, as no information is given in the text regarding the specifics of the implementation.

```
class DivineFavor(SpellCard):
    def __init__(self):
        super().__init__("Divine Favor", 3,
    CHARACTER_CLASS.PALADIN, CARD_RARITY.RARE)
    def use(self, player, game):
        super().use(player, game)
        difference = len(game.other_player.hand)
       - len(player.hand)
        for i in range(0, difference):
            player.draw()
```

Figure 2: Code for the HS card "Divine Favor".

3 Problem Definition

Given the description of a card x, our decoding problem is to find the code $\hat{y}$ so that:

$$\hat{y} = \underset{y}{\operatorname{argmax}} \log P(y \mid x) \tag{1}$$

Here $\log P(y \mid x)$ is estimated by a given model. We define $y = y_1..y_{|y|}$ as the sequence of characters of the code with length $|y|$. We index each input field with $k = 1..|x|$, where $|x|$ quantifies the number of input fields. $|x_k|$ denotes the number of tokens in x_k and x_{ki} selects the i-th token.

4 Structured Attention

Background When $|x| = 1$, the attention model of [7] applies. Following the chain rule, $\log P(y|x) = \sum_{t=1..|y|} \log P(y_t|y_1..y_{t-1}, x)$, each token y_t is predicted conditioned on the previously generated sequence $y_1..y_{t-1}$ and input sequence $x_1 = x_{11}..x_{1|x_1|}$. Probability are estimated with a softmax over the vocabulary Y:

$$p(y_t|y_1..y_{t-1}, x_1) = \underset{y_t \in Y}{\operatorname{softmax}}(\mathbf{h}_t) \tag{2}$$

where $\mathbf{h}_t$ is the Recurrent Neural Network (RNN) state at time stamp t, which is modeled as $g(\mathbf{y}_{t-1}, \mathbf{h}_{t-1}, \mathbf{z}_t)$. $g(\cdot)$ is a recurrent update function for generating the new state $\mathbf{h}_t$ based on the previous token $\mathbf{y}_{t-1}$, the previous state $\mathbf{h}_{t-1}$, and the input text representation $\mathbf{z}_t$. We implement g using a Long Short-Term Memory (LSTM) RNNs [8].

The attention mechanism generates the representation of the input sequence $\mathbf{x} = x_{11}..x_{1|x_1|}$, and $\mathbf{z}_t$ is computed as the weighted sum $\mathbf{z}_t = \sum_{i=1..|x_1|} a_i h(x_{1i})$, where a_i is the attention coefficient

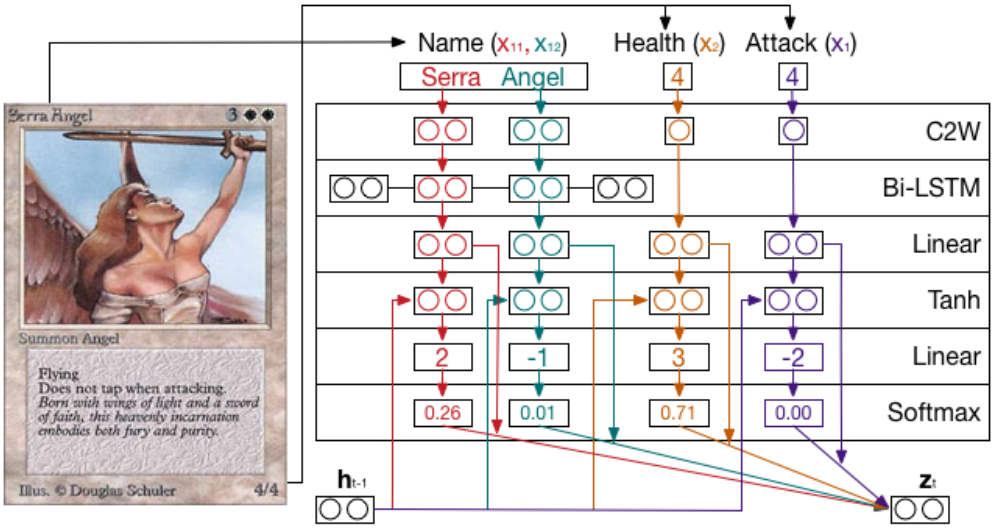

Figure 3: Illustration of the structured attention mechanism operating on a single time stamp t.

obtained for token x_{1i} and h is a function that maps each x_{1i} to a continuous vector. In general, h is a function that projects x_{1i} by learning a lookup table, and then embedding contextual words by defining an RNN. Coefficients a_i are computed with a softmax over input tokens $x_{11}..x_{1|x_1|}$:

$$a_i = \underset{x_{1i} \in x}{\text{softmax}} \left(v(h(x_{1i}), \mathbf{h}_{t-1}) \right) \tag{3}$$

Function v computes the affinity of each token x_{1i} and the current output context $\mathbf{h}_{t-1}$. A common implementation of v is to apply a linear projection from $h(x_{1i}) : \mathbf{h}_{t-1}$ (where : is the concatenation operation) into a fixed size vector, followed by a $\tanh$ and another linear projection.

Our Approach We extend the computation of $\mathbf{z}_t$ for cases when x corresponds to multiple fields. Figure 3 illustrates how the MTG card "Serra Angel" is encoded, assuming that there are two singular fields and one text field. We first encode each token x_{ki} using the C2W model described in [9], which is a replacement for lookup tables where word representations are learned at the character level (cf. *C2W* row). A context-aware representation is built for words in the text fields using a bidirectional LSTM (cf. *Bi-LSTM* row). Computing attention over multiple input fields is problematic as each input field's vectors have different sizes and value ranges. Thus, we learn a linear projection mapping each input token x_{ki} to a vector with a common dimensionality and value range (cf. *Linear* row). Denoting this process as $f(x_{ki})$, we extend Equation 3 as:

$$a_{ki} = \underset{x_{ki} \in x}{\text{softmax}}(v(f(x_{ki}), \mathbf{h}_{t-1})) \tag{4}$$

Here a scalar coefficient a_{ki} is computed for each input token x_{ki} (cf. "Tanh", "Linear", and "Softmax" rows). Thus, the overall input representation $\mathbf{z}_t$ is computed as:

$$\mathbf{z}_t = \sum_{k=1..|x|, i=1..|x_k|} a_{ij} f(x_{ki}) \tag{5}$$

5 Latent Predictor Networks

Background In order to decode from x to y, many words must be copied into the code, such as the name of the card, the attack and the cost values. If we observe the HS card in Figure 1 and the respective code in Figure 2, we observe that the name "Divine Favor" must be copied into the class name and in the constructor, along with the cost of the card "3". As explained earlier, this problem is not specific to our task: for instance, in the dataset of [10], a model must learn to map from `timeout = int ( timeout )` to "convert timeout into an integer.", where the name of

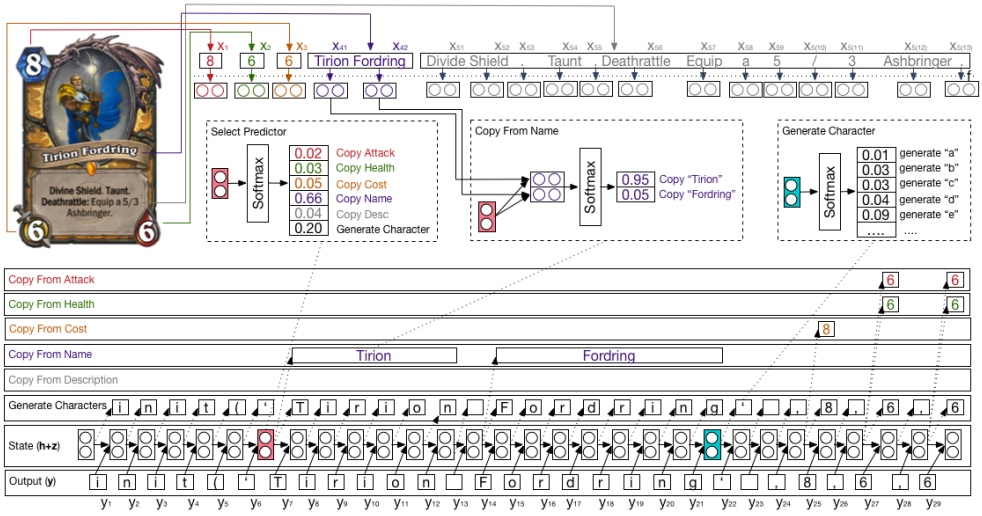

Figure 4: Generation process for the code `init('Tirion Fordring',8,6,6)` using LPNs.

the variable "timeout" must be copied into the output sequence. The same issue exists for proper nouns in machine translation which are typically copied from one language to the other. Pointer networks [11] address this by defining a probability distribution over a set of units that can be copied $c = c_1..c_{|c|}$. The probability of copying a unit c_i is modeled as:

$$p(c_i) = \underset{c_i \in c}{\text{softmax}}(v(h(c_i), \mathbf{q})) \tag{6}$$

As in the attention model (Equation 3), v is a function that computes the affinity between an embedded copyable unit $h(c_i)$ and an arbitrary vector $\mathbf{q}$.

Our Approach Combining pointer networks with a character-based softmax is in itself difficult as these generate segments of different granularity and there is no ground truth of which predictor to use at each time stamp. We now describe Latent Predictor Networks, which model the conditional probability $\log P(y|x)$ over the latent sequence of predictors used to generate y.

We assume that our model uses multiple predictors $r \in R$, where each r can generate multiple segments $s_t = y_t..y_{t+|s_t|-1}$ with arbitrary length $|s_t|$ at time stamp t. An example is illustrated in Figure 4, where we observe that to generate the code `init('Tirion Fordring',8,6,6)`, a pointer network can be used to generate the sequences y_7^{13}=`Tirion` and y_{14}^{22}=`Fordring` (cf. "Copy From Name" row). These sequences can also be generated using a character softmax (cf. "Generate Characters" row). The same applies to the generation of the attack, health and cost values as each of these predictors is an element in R. Thus, we define our objective function as a marginal log likelihood function over a latent variable ω:

$$\log P(y \mid x) = \log \sum_{\omega \in \bar{\omega}} P(y, \omega \mid x) \tag{7}$$

Formally, ω is a sequence of pairs r_t, s_t, where $r_t \in R$ denotes the predictor that is used at timestamp t and s_t the generated string. We decompose $P(y, \omega \mid x)$ as the product of the probabilities of segments s_t and predictors r_t:

$$P(y, \omega \mid x) = \prod_{r_t, s_t \in \omega} P(s_t, r_t \mid y_1..y_{t-1}, x) =$$

$$\prod_{r_t, s_t \in \omega} P(s_t \mid y_1..y_{t-1}, x, r_t) P(r_t \mid y_1..y_{t-1}, x) \tag{8}$$

where the generation of each segment is performed in two steps: select the predictor r_t with probability $P(r_t \mid y_1..y_{t-1}, x)$ and then generate s_t conditioned on predictor r_t with probability

$\log P(s_t \mid y_1..y_{t-1}, x, r_t)$. The probability of each predictor is computed using a softmax over all predictors in R conditioned on the previous state $\mathbf{h_{t-1}}$ and the input representation $\mathbf{z}_t$ (cf. "Select Predictor" box). Then, the probability of generating the segment s_t depends on the predictor type. We define three types of predictors:

Character Generation Generate a single character from observed characters from the training data. Only one character is generated at each time stamp with probability given by Equation 2.

Copy Singular Field For singular fields only the field itself can be copied, for instance, the value of the attack and cost attributes or the type of card. The size of the generated segment is the number of characters in the copied field and the segment is generated with probability 1.

Copy Text Field For text fields, we allow each of the words x_{ki} within the field to be copied. The probability of copying a word is learned with a pointer network (cf. "Copy From Name" box), where $h(c_i)$ is set to the representation of the word $f(x_{ki})$ and $\mathbf{q}$ is the concatenation $\mathbf{h}_{t-1} : \mathbf{z}_t$ of the state and input vectors. This predictor generates a segment with the size of the copied word.

It is important to note that the state vector $\mathbf{h}_{t-1}$ is generated by building an RNN over the sequence of characters up until the time stamp $t-1$, i.e. the previous context $\mathbf{y}_{t-1}$ is encoded at the character level. This allows the number of possible states to remain tractable at training time.

5.1 Inference

At training time we use back-propagation to maximize the probability of observed code, according to Equation 7. Gradient computation must be performed with respect to each computed probability $P(r_t \mid y_1..y_{t-1}, x)$ and $P(s_t \mid y_1..y_{t-1}, x, r_t)$. The derivative $\frac{\partial \log P(y|x)}{\partial P(r_t|y_1..y_{t-1},x)}$ yields:

$$\frac{\partial \alpha_t P(r_t \mid y_1..y_{t-1}, x)\beta_{t,r_t} + \xi_{r_t}}{P(y \mid x)\partial P(r_t \mid y_1..y_{t-1}, x)} = \frac{\alpha_t \beta_{t,r_t}}{\alpha_{|y|+1}} \tag{9}$$

Here α_t denotes the cumulative probability of all values of ω up until time stamp t and β_{t,r_t} denotes the cumulative probability starting from predictor r_t at time stamp t, exclusive. $\alpha_{|y|+1}$ yields the marginal probability $P(y \mid x)$. For completeness, ξ_r denotes the cumulative probabilities of all ω that do not include r_t. α and β can be computed efficiently using the forward-backward algorithm for Semi-Markov models [12], where we associate $P(r_t \mid y_1..y_{t-1}, x)$ to edges and $P(s_t \mid y_1..y_{t-1}, x, r_t)$ to nodes in the Markov chain. The derivative $\frac{\partial \log P(y|x)}{\partial P(s_t|y_1..y_{t-1},x,r_t)}$ can be computed using the same logic.

5.2 Decoding

Decoding is performed using a stack-based decoder with beam search. Each state S corresponds to a choice of predictor r_t and segment s_t at a given time stamp t. This state is scored as $V(S) = \log P(s_t \mid y_1..y_{t-1}, x, r_t) + \log P(r_t \mid y_1..y_{t-1}, x) + V(prev(S))$, where $prev(S)$ denotes the predecessor state of S. At each time stamp, the n states with the highest scores V are expanded, where n is the size of the beam. For each predictor r_t, each output s_t generates a new state. Finally, at each timestamp t, all states which produce the same output up to that point are merged by summing their probabilities.

6 Code Compression

As the attention-based model traverses all input units at each generation step, generation becomes quite expensive for datasets such as MTG where the average card code contains 1,080 characters. While this is not the essential contribution in our paper, we propose a simple method to compress the code while maintaining the structure of the code, allowing us to train on datasets with longer code (e.g., MTG).

The idea behind that method is that many keywords in the programming language (e.g., `public` and `return`) as well as frequently used functions and classes (e.g., `Card`) can be learned without

X	v	size
X_1	card)⇓{⇓super(card);⇓}⇓@Override⇓public	1041
X_2	bility	1002
X_3	;⇓this.	964
X_4	(UUID_ownerId)⇓{⇓super(ownerId	934
X_5	public_	907
X_6	new_	881
X_7	_copy()	859
X_8	}")X_3expansionSetCode_=_"	837
X_9	X_6CardType[]{CardType.	815
X_{10}	ffect	794

Table 2: First 10 compressed units in MTG. We replaced newlines with ⇓ and spaces with _.

character level information. We exploit this by mapping such strings onto additional symbols X_i (e.g., `public class copy()` → "X_1 X_2 X_3()"). Formally, we seek the string $\hat{v}$ among all strings $V(max)$ up to length max that maximally reduces the size of the corpus:

$$\hat{v} = \underset{v \in V(max)}{\text{argmax}} \; (len(v) - 1)C(v) \tag{10}$$

where $C(v)$ is the number of occurrences of v in the training corpus and $len(v)$ its length. $(len(v) - 1)C(v)$ can be seen as the number of characters reduced by replacing v with a non-terminal symbol. To find $q(v)$ efficiently, we leverage the fact that $C(v) \leq C(v')$ if v contains v'. It follows that $(max - 1)C(v) \leq (max - 1)C(v')$, which means that the maximum compression obtainable for v at size max is always lower than that of v'. Thus, if we can find a $\bar{v}$ such that $(len(\bar{v}) - 1)C(\bar{v}) > (max - 1)C(v')$, that is $\bar{v}$ at the current size achieves a better compression rate than v' at the maximum length, then it follows that all sequences that contain v can be discarded as candidates. Based on this idea, our iterative search starts by obtaining the counts $C(v)$ for all segments of size $s = 2$, and computing the best scoring segment $\bar{v}$. Then, we build a list $L(s)$ of all segments that achieve a better compression rate than $\bar{v}$ at their maximum size. At size $s + 1$, only segments that contain a element in $L(s - 1)$ need to be considered, making the number of substrings to be tested to be tractable as s increases. The algorithm stops once s reaches max or the newly generated list $L(s)$ contains no elements.

Once $\hat{v}$ is obtained, we replace all occurrences of $\hat{v}$ with a new non-terminal symbol. This process is repeated until a desired average size for the code is reached. While training is performed on the compressed code, the decoding will undergo an additional step, where the compressed code is restored by expanding the all X_i. Table 2 shows the first 10 replacements from the MTG dataset, reducing its average size from 1080 to 794.

7 Experiments

Datasets Tests are performed on the two datasets provided in this paper, described in Table 1. Additionally, to test the model's ability of generalize to other domains, we report results in the Django dataset [10], comprising of 16000 training, 1000 development and 1805 test annotations. Each data point consists of a line of Python code together with a manually created natural language description.

Neural Benchmarks We implement two standard neural networks, namely a sequence-to-sequence model [13] and an attention-based model [7]. The former is adapted to work with multiple input fields by concatenating them, while the latter uses our proposed attention model. These models are denoted as "Sequence" and "Attention".

Machine Translation Baselines Our problem can also be viewed in the framework of semantic parsing [2, 14, 3, 5]. Unfortunately, these approaches define strong assumptions regarding the grammar and structure of the output, which makes it difficult to generalize for other domains [15]. However, the work in [16] provides evidence that using machine translation systems without committing

to such assumptions can lead to results competitive with the systems described above. We follow the same approach and create a phrase-based [1] model and a hierarchical model (or PCFG) [17] as benchmarks for the work presented here. As these models are optimized to generate words, not characters, we implement a tokenizer that splits on all punctuation characters, except for the "_" character. We also facilitate the task by splitting CamelCase words (e.g., `class TirionFordring` $\rightarrow$ `class Tirion Fordring`). Otherwise all class names would not be generated correctly by these methods. We used the models implemented in Moses to generate these baselines using standard parameters, using IBM Alignment Model 4 for word alignments [18], MERT for tuning [19] and a 4-gram Kenser-Ney Smoothed language model [20]. These models will be denoted as "Phrase" and "Hierarchical", respectively.

Retrieval Baseline It was reported in [6] that a simple retrieval method that outputs the most similar input for each sample, measured using Levenshtein Distance, leads to good results. We implement this baseline by computing the average Levenshtein Distance for each input field. This baseline is denoted "Retrieval".

Evaluation A typical metric is to compute the accuracy of whether the generated code exactly matches the reference code. This is informative as it gives an intuition of how many samples can be used without further human post-editing. However, it does not provide an illustration on the degree of closeness to achieving the correct code. Thus, we also test using BLEU-4 [21] at the token level.

Setup The multiple input types (Figure 3) are hyper-parametrized as follows: The C2W model (cf. "C2W" row) used to obtain continuous vectors for word types uses character embeddings of size 100 and LSTM states of size 300, and generates vectors of size 300. We also report on results using word lookup tables of size 300, where we replace singletons with a special unknown token with probability 0.5 during training, which is then used for out-of-vocabulary words. For text fields, the context (cf. "Bi-LSTM" row) is encoded with a Bi-LSTM of size 300 for the forward and backward states. Finally, a linear layer maps the different input tokens into a common space with of size 300 (cf. "Linear" row). As for the attention model, we used an hidden layer of size 200 before applying the non-linearity (row "Tanh"). As for the decoder (Figure 4), we encode output characters with size 100 (cf. "output (y)" row), and an LSTM state of size 300 and an input representation of size 300 (cf. "State(h+z)" row). For each pointer network (e.g., "Copy From Name" box), the intersection between the input units and the state units are performed with a vector of size 200. Training is performed using mini-batches of 20 samples using AdaDelta [22] and we report results using the iteration with the highest BLEU score on the validation set (tested at intervals of 5000 mini-batches). Decoding is performed with a beam of 1000. As for compression, we performed a grid search over compressing the code from 0% to 80% of the original average length over intervals of 20% for the HS and Django datasets. On the MTG dataset, we are forced to compress the code up to 80% due to performance issues when training with extremely long sequences.

7.1 Results

Baseline Comparison Results are reported in Table 3. Regarding the retrieval results (cf. "Retrieval" row), we observe the best BLEU scores among the baselines in the card datasets (cf. "MTG" and "HS" columns). A key advantage of this method is that retrieving existing entities guarantees that the output is well formed, with no syntactic errors such as producing a non-existent function call or generating incomplete code. As BLEU penalizes length mismatches, generating code that matches the length of the reference provides a large boost. The phrase-based translation model (cf. "Phrase" row) performs well in the Django (cf. "Django" column), where mapping from the input to the output is mostly monotonic, while the hierarchical model (cf. "Hierarchical" row) yields better performance on the card datasets as the concatenation of the input fields needs to be reordered extensively into the output sequence. Finally, the sequence-to-sequence model (cf. "Sequence" row) yields extremely low results, mainly due to the lack of capacity needed to memorize whole input and output sequences, while the attention based model (cf. "Attention" row) produces results on par with phrase-based systems. Finally, we observe that by including all the proposed components (cf. "Our System" row), we obtain significant improvements over all baselines in the three datasets and is the only one that obtains non-zero accuracies in the card datasets.

	MTG		HS		Django	
	BLEU	Acc	BLEU	Acc	BLEU	Acc
Retrieval	54.9	0.0	62.5	0.0	18.6	14.7
Phrase	49.5	0.0	34.1	0.0	47.6	31.5
Hierachical	50.6	0.0	43.2	0.0	35.9	9.5
Sequence	33.8	0.0	28.5	0.0	44.1	33.2
Attention	50.1	0.0	43.9	0.0	58.9	38.8
Our System	**61.4**	**4.8**	65.6	4.5	**77.6**	**62.3**
– C2W	60.9	4.4	**67.1**	4.5	75.9	60.9
– Compress	-	-	59.7	**6.1**	76.3	61.3
– LPN	52.4	0.0	42.0	0.0	63.3	40.8
– Attention	39.1	0.5	49.9	3.0	48.8	34.5

Table 3: BLEU and Accuracy scores for the proposed task on two in-domain datasets (HS and MTG) and an out-of-domain dataset (Django).

Compression	0%	20%	40%	60%	80%
Seconds Per Card					
Softmax	2.81	2.36	1.88	1.42	**0.94**
LPN	3.29	2.65	2.35	1.93	**1.41**
BLEU Scores					
Softmax	44.2	46.9	47.2	51.4	**52.7**
LPN	59.7	62.8	61.1	66.4	**67.1**

Table 4: Results with increasing compression rates with a regular softmax (cf. "Softmax") and a LPN (cf. "LPN"). Performance values (cf. "Seconds Per Card" block) are computed using one CPU.

Component Comparison We present ablation results in order to analyze the contribution of each of our modifications. Removing the C2W model (cf. "– C2W" row) yields a small deterioration, as word lookup tables are more susceptible to sparsity. The only exception is in the HS dataset, where lookup tables perform better. We believe that this is because the small size of the training set does not provide enough evidence for the character model to scale to unknown words. Surprisingly, running our model compression code (cf. "– Compress" row) actually yields worse results. Table 4 provides an illustration of the results for different compression rates. We obtain the best results with an 80% compression rate (cf. "BLEU Scores" block), while maximising the time each card is processed (cf. "Seconds Per Card" block). While the reason for this is uncertain, it is similar to the finding that language models that output characters tend to under-perform those that output words [23]. This applies when using the regular optimization process with a character softmax (cf. "Softmax" rows), but also when using the LPN (cf. "LPN" rows). We also note that the training speed of LPNs is not significantly lower as marginalization is performed with a dynamic program. Finally, a significant decrease is observed if we remove the pointer networks (cf. "– LPN" row). These improvements also generalize to sequence-to-sequence models (cf. "– Attention" row), as the scores are superior to the sequence-to-sequence benchmark (cf. "Sequence" row).

Result Analysis Examples of the code generated for two cards are illustrated in Figure 5. We obtain the segments that were copied by the pointer networks by computing the most likely predictor for those segments. We observe from the marked segments that the model effectively copies the attributes that match in the output, including the name of the card that must be collapsed. As expected, the majority of the errors originate from inaccuracies in the generation of the effect of the card. While it is encouraging to observe that a small percentage of the cards are generated correctly, it is worth mentioning that these are the result of many cards possessing similar effects. The "Madder Bomber" card is generated correctly as there is a similar card "Mad Bomber" in the training set, which implements the same effect, except that it deals 3 damage instead of 6. Yet, it is a promising result that the model was able to capture this difference. However, in many cases, effects that radi-

cally differ from seen ones tend to be generated incorrectly. In the card "Preparation", we observe that while the properties of the card are generated correctly, the effect implements a unrelated one, with the exception of the value 3, which is correctly copied. Yet, interestingly, it still generates a valid effect, which sets a minion's attack to 3. Investigating better methods to accurately generate these effects will be object of further studies.


```
class MadderBomber(MinionCard):        BLEU = 100.0
  def __init__(self):
    super().__init__("Madder Bomber", 5,
    CHARACTER_CLASS.ALL, CARD_RARITY.RARE,
    battlecry=Battlecry(Damage(1),
    CharacterSelector(players=BothPlayer(),
       picker= RandomPicker(6))))

def create_minion(self, player):§
  return Minion(5, 4)§
```


```
class Preparation(SpellCard):                BLEU = 64.2
  def __init__(self):
    super().__init__("Preparation", 0,
    CHARACTER_CLASS.ROGUE, CARD_RARITY.EPIC,
    target_func=hearthbreaker.targeting.find_minion_spell_target)

  def use(self, player, game):
    super().use(player, game)
    self.target.change_attack(3)
                          player.add_aura(AuraUntil(ManaChange(-3),
                          CardSelector(condition=IsSpell()), SpellCast()))
```

Figure 5: Examples of decoded cards from HS. Copied segments are marked in green and incorrect segments are marked in red.

8 Related Work

While we target widely used programming languages, namely, Java and Python, our work is related to studies on the generation of any executable code. These include generating regular expressions [24], and the code for parsing input documents [4]. Much research has also been invested in generating formal languages, such as database queries [25, 26], agent specific language [27] or smart phone instructions [28]. Finally, mapping natural language into a sequence of actions for the generation of executable code [29]. Finally, a considerable effort in this task has focused on semantic parsing [2, 3, 4, 5, 6]. Recently proposed models focus on Combinatory Categorical Grammars [24, 5], Bayesian Tree Transducers [3, 4] and Probabilistic Context Free Grammars [16]. The work in natural language programming [30, 31], where users write lines of code from natural language, is also related to our work. Finally, the reverse mapping from code into natural language is explored in [10].

Character-based sequence-to-sequence models have previously been used to generate code from natural language in [32]. Inspired by these works, LPNs provide a richer framework by employing attention models [7], pointer networks [11] and character-based embeddings [9]. Our formulation can also be seen as a generalization of [33], who implement a special case where two predictors have the same granularity (a sub-token softmax and a pointer network). Finally, HMMs have been employed in neural models to marginalize over label sequences in [34, 35] by modeling transitions between labels.

9 Conclusion

We introduced a neural network architecture named *Latent Prediction Network*, which allows efficient marginalization over multiple predictors. Under this architecture, we propose a generative model for code generation that combines a character level softmax to generate language-specific tokens and multiple pointer networks to copy keywords from the input. Along with other extensions, namely structured attention and code compression, our model is applied on on both existing datasets and also on a newly created one with implementations of TCG game cards. Our experiments show that our model out-performs multiple benchmarks, which demonstrate the importance of combining different types of predictors.

References

[1] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, pages 177–180, 2007.

[2] Yuk Wah Wong and Raymond J. Mooney. Learning for semantic parsing with statistical machine translation. In *Proceedings of the Main Conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pages 439–446, 2006.

[3] Bevan Keeley Jones, Mark Johnson, and Sharon Goldwater. Semantic parsing with bayesian tree transducers. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*, pages 488–496, 2012.

[4] Tao Lei, Fan Long, Regina Barzilay, and Martin Rinard. From natural language specifications to program input parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1294–1303, Sofia, Bulgaria, August 2013.

[5] Yoav Artzi, Kenton Lee, and Luke Zettlemoyer. Broad-coverage ccg semantic parsing with amr. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1699–1710, September 2015.

[6] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics*, pages 878–888, Beijing, China, July 2015.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

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

[9] Wang Ling, Tiago Luís, Luís Marujo, Rámon Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan W Black, and Isabel Trancoso. Finding function in form: Compositional character models for open vocabulary word representation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015.

[10] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, Nebraska, USA, November 2015.

[11] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In C. Cortes, N.D. Lawrence, D.D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2674–2682. Curran Associates, Inc., 2015.

[12] Sunita Sarawagi and William W. Cohen. Semi-markov conditional random fields for information extraction. In L. K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 1185–1192. MIT Press, 2005.

[13] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[14] Wei Lu, Hwee Tou Ng, Wee Sun Lee, and Luke S. Zettlemoyer. A generative model for parsing natural language to meaning representations. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 783–792, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[15] Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. Inducing probabilistic ccg grammars from logical form with higher-order unification. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1223–1233, 2010.

[16] Jacob Andreas, Andreas Vlachos, and Stephen Clark. Semantic parsing as machine translation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 47–52, August 2013.

[17] David Chiang. Hierarchical phrase-based translation. *Comput. Linguist.*, 33(2):201–228, June 2007.

[18] Franz Josef Och and Hermann Ney. A systematic comparison of various statistical alignment models. *Comput. Linguist.*, 29(1):19–51, March 2003.

[19] Artem Sokolov and François Yvon. Minimum Error Rate Semi-Ring. In Mikel Forcada and Heidi Depraetere, editors, *Proceedings of the European Conference on Machine Translation*, pages 241–248, Leuven, Belgium, 2011.

[20] Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H. Clark, and Philipp Koehn. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51th Annual Meeting on Association for Computational Linguistics*, pages 690–696, 2013.

[21] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 311–318, 2002.

[22] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[23] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

[24] Nate Kushman and Regina Barzilay. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 826–836, Atlanta, Georgia, June 2013.

[25] John M. Zelle and Raymond J. Mooney. Learning to parse database queries using inductive logic programming. In *AAAI/IAAI*, pages 1050–1055, Portland, OR, August 1996. AAAI Press/MIT Press.

[26] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, 2013.

[27] Rohit J. Kate, Yuk Wah Wong, and Raymond J. Mooney. Learning to transform natural to formal languages. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 1062–1068, Pittsburgh, PA, July 2005.

[28] Vu Le, Sumit Gulwani, and Zhendong Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, pages 193–206, 2013.

[29] S. R. K. Branavan, Harr Chen, Luke S. Zettlemoyer, and Regina Barzilay. Reinforcement learning for mapping instructions to actions. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 82–90, 2009.

[30] David Vadas and James R. Curran. Programming with unrestricted natural language. In *Proceedings of the Australasian Language Technology Workshop 2005*, pages 191–199, Sydney, Australia, December 2005.

[31] Mehdi Hafezi Manshadi, Daniel Gildea, and James F. Allen. Integrating programming by example and natural language programming. In Marie desJardins and Michael L. Littman, editors, *AAAI*. AAAI Press, 2013.

[32] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. On end-to-end program generation from user intention by deep neural networks. *CoRR*, abs/1510.07211, 2015.

[33] M. Allamanis, H. Peng, and C. Sutton. A Convolutional Attention Network for Extreme Summarization of Source Code. *ArXiv e-prints*, February 2016.

[34] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, November 2011.

[35] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer. Neural Architectures for Named Entity Recognition. *ArXiv e-prints*, March 2016.