Generating Latex Code for Commutative Diagrams

Wenqi Li
Stanford University
wenqili@stanford.edu

Abstract

We tackle the task of generating Latex code for a given commutative diagram using a encoder-decoder model paradigm. This is a novel application, and we managed to generate compilable Latex codes that produces reasonable diagrams using a CNN encoder and a transformer decoder.

1 Introduction

This project aims to train a model that generates latex code when given the image of a commutative diagram, which is a graphical presentations of objects and maps and is commonly used in mathematical papers. See Figure 1 for an example of a commutative diagram. Commutative diagrams are most

$$\begin{array}{ccc} \mathcal{F}' & \xrightarrow[s']{} & f^{-1}\mathcal{G}' \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle f^{-1}b} \\ \mathcal{F} & \xrightarrow{s} & f^{-1}\mathcal{G} \end{array}$$

Figure 1: An example of a commutative diagram

commonly typeset using latex with the package `tikzcd` as `tikzcd`s or `xypic` as `xymatrix`s. One possible version of latex code for Figure 1 is

```
\xymatrix{
\mathcal{F}' \ar[r]_-{s'} \ar[d]_a &
f^{-1}\mathcal{G}' \ar[d]^{f^{-1}b} \\
\mathcal{F} \ar[r]^-s &
f^{-1}\mathcal{G}
}
```

In general, a commutative diagram can have any number of nodes and any number of arrows with labels. Typesetting a commutative diagram can be cumbersome, especially when the diagram becomes complicated. Very often one needs to read a mathematical paper (in PDF format) and reproduce the commutative diagrams in it for one's own use, so an application that can convert images of commutative diagrams to latex code can be very helpful for researchers. This project will try to apply machine learning methods to this task.

We note that this problem on its own, ignoring the real-world application side, is an interesting image to sequence generation task. In particular, almost all of the pixels of images of commutative diagrams are white, and only a small amount of pixels are black, tracing out the diagram. So the actual information is sparse in the image. This is very different from usual images, say of a cat, where the information is dense in the image. On the output side, we are trying to output latex code, in contrast to words and sentences in usual NLP tasks. It is a challenge to output latex codes that can compile.

CS230 2022, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

2 Related Work

The task of generating Latex code given an image of a commutative diagram is not found in the literature. However, the image captioning task is most similar to our code generation task at hand, because in both cases the inputs are images and the outputs are strings. In 2015, Vinyals et al. [1]'s work gave a neural network method that produces a description of a given image, achieving reasonable BLEU scores on varioius dataset. In their work, they used a LSTM-based sentence generator, which we will adapt as our baseline model. Later, Xu et al. [2] developed an image captioning model using hard and soft stochastic attention mechanisms. In their model, they used a convolutional neural network as an encoder, and a LSTM network with attention as a decoder. We will adapt the idea of using a CNN as an encoder in our models. Although the image captioning task has the same input/output format as our code generation task, we note the following differences: the image captioning task aims to extract the semantic meaning contained in an image, and express that meaning using natural language, whereas our tasks requires the model to output the code that produces the exact same diagram. Moreover, since we cannot operate on the level of words (there are no "words" in Latex), we must operate mostly on the character level, so the length of sequences for our task will be significantly longer. This will make the gradient vanishing problem for recurrent networks more severe, even if the LSTM cells can help ameliorate it.

In 2017, Vaswani et al. [3] published the seminal paper "Attention is All You Need", where they proposed the transformer architecture. The transformer replies solely on attention mechanisms, therefore replacing the recurrent networks as a decoder. This helps solve the vanishing gradient problem even in our task, so we adapt it as another baseline model, and attempt to improve on it. On the encoder side, we seek convolutional neural networks that can serve as feature extracters. The VGG network developed by Simonyan and Zisserman [4] is a powerful feature extracter for images, as it is trained on ImageNet and achieved state-of-the-art results in the ImageNet Challenge. The ResNet developed by He et al. [5] is another deep CNN, solving the problem of very deep networks being diffcult to train by adding skip connections. We try both of these networks as encoders for our task.

3 Dataset

We first describe the data. All data are collected from the Stacks Project, which is an online open source textbook for algebraic geometry. We downloaded all the latex files of this text, and parsed out all pieces of codes that are used to generate commutative diagrams. There are 4345 commutative diagrams coded in the Stacks Project in total, and we parsed out all of them. To obtain the images, we put each code block for one diagram inside one latex document, and compile all such documents as `standalone` documents. To reduce computational cost, we choose a resolution of 270×200. Therefore we obtain 4345 pairs of diagram images and latex code. We randomly shuffle the 4345 pairs, and split into 4000 training examples, 175 development examples, and 170 test examples. Note that the shuffling is important since in an algebraic geometry text book, near by diagrams are likely to be similarly. The shuffling helps to ensure the training, development, and test data are from roughly the same distribution.

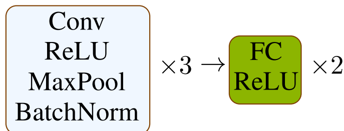
For a typical example in the dataset, see Figure 1 and the code below it.

4 Method

All models we experiment with can be described as encoder-decoder models. The models consist of an encoder network, which takes the images of commutative diagrams as inputs, and outputs a feature vector. The decoder takes this feature vector as input, and outputs a probability distribution over the set of all possible tokens. The latex code is then generated by sampling from this probability distribution one token at a time, until the <END> token is reached or the maximum allowed length is reached. Here we describe respectively what encoders and decoders we have experimented, and the what combinations of them are used will be discussed in the Experiments section.

First of all we have the trivial encoder: the encoder simply flattens the image, and uses that as the feature vector. This together with the decoders explained below consist of our baselines. Besides the trivial encoder, we used a variety of convolutional neural networks as the encoder. The first one of

these is a customized CNN that we designed “by hand”, with the following architecture:



where “Conv” is a 2D convolutional layer, “FC” is a fully connected layer, and $\times n$ means the previous blocked combination of layers is repeated n times in that order.

We also tried to use pretrained CNNs as encoders. We used the VGG network and the ResNet, both with weights pretrained on ImageNet and with no pretrained weights.

For decoders, we tried a LSTM network and a transformer decoder. The LSTM network is a recurrent neural network that uses hidden cells consisting of several gates. A detailed explanation can be found in [6]. The Transformer decoder has several repeated layers, each of which consists of a self-attention layer, a multihead attention layer, and a fully connected layer, with LayerNorm layers in between. At each “LayerNorm” layer, we also add the input of the previous layer, similar to residual networks. The multihead attention layer receives input both from the previous layer and from the output of the encoder. The number of these blocks used in the decoder is a hyper parameter. After each attention layer, there is also a dropout layer, but we omit it in the graphical presentation since it is an important part of the architecture. For more details, see [3].

For Latex codes, we do not have well-defined “words”, so for both decoders, we use two embedding methods other than usual word embeddings. The first one is character embedding, where the set of all “vocabulary” is the set of all lower and upper case letters, and all characters that appear in Latex codes. The embedding from “vocabulary” index to vectors is learned during training. The second embedding methods has a “vocabulary” that includes all characters as in the previous method, and also latex commands commonly seen in the dataset, such as `times`, `otimes`, `beta`, `gamma`, `delta`, etc.

In all models, we use a temporal cross entropy loss as our loss function. This is computed for an output tensor $\hat{y}$ which contains class scores (here a class is a token in the vocabulary) for all time steps (here a time step is a position in a Latex code string), and a true label y giving the true class at each time step. The loss is computed by taking the cross entropy loss at each time step, and summing up across all time steps. We note that to facilitate batched training, each output sequence are padded with <NULL> tokens so that they have the same length. The loss computed at time steps where the true label is <NULL> is ignored. To evaluate the performance, we use the BLEU score [7]. Here, for each image, there is only one each model-generated candidate sequence, and only one reference sequence, which is the true code. The BLEU score is computed by multiplying a brevity penalty and an exponentially averaged n-gram precision up to a selected n_{max}, the maximum gram number. Here we use $n_{\text{max}} = 4$.

Lastly, the predicted sequence is generated by sampling from the decoder output one time step at a time. We use two different sampling methods. The first method is taking the token that has the maximum probability as predicted by the decoder. The raw output of the decoder are logits, but a maximum value in terms of logits corresponds to a maximum probability, so the sampled token is the one with the maximal output value. The second is method is to sample tokens randomly according to the probability distribution predicted by the decoder. We first convert the logits into probability using the softmax function, and sample the next token randomly according to this predicted probability distribution. Compared to maximum sampling, this method prevents the model from keep outputting the same token (e.g. the white space character).

5 Experiments

We summarize the representative experiments we did in the following table:

In Table 1, the encoder “CNN” refers to the custom CNN described above. The “vocab” in the Tokens column means a vocabulary consisting of all characters together with common Latex commands is used. In all experiments, the optimizer is Adam and the initial learning rate is fixed to be 0.001. We performed a log-scale grid hyperparameter search for the learning rate and the weight decay factor: for learning rates, 3e-3, 1e-3, 3e-4, 1e-4, 3e-5 were experimented, and for the weight decay factor

Trials	Encoder	Decoder	Weight decay	Decoder Layers	Sampling	Tokens
1	Trivial	LSTM	0		max	characters
2	Trivial	Transformer	0	2	max	characters
3	CNN	Transformer	3e-6	2	max	characters
4	CNN	Transformer	3e-6	3	max	characters
5	CNN	Transformer	3e-6	3	max	vocab
6	CNN	Transformer	3e-6	3	random	vocab
7	ResNet	Transformer	3e-6	2	max	characters
8	VGG	Transformer	3e-6	2	max	characters

Table 1: Models and Hyperparameters

1e-6, 3e-6, 1e-7 were experimented. A 3e-3 learning rate or a 1e-6 weight decay made the model not converging at all (training loss increased instead of decreasing), and a learning rate lower than 3e-4 resulted in slow training. The choice between an initial learning rate of 1e-3 and 3e-4 and the choice between a decay factor of 3e-6 and 1e-7 made no significant difference. To facilitate training, we used learning rate decay. For the VGG and ResNet encoders, we tried to use the pretrained weights obtained by training on ImageNet, but they both give BLEU scores lower than 0.1 on both train and dev sets, and the model could not output any meaningful predictions.

We fixed the batch size to be 64, which is the maximum allowed batch size by the memory constraint of the machine. In all experiments with the transformer decoder, we used 4 attention heads, which is also the maximum allowed by the memory constraints. For the custom CNN encoder, the three convolutional layers have 5×5 kernels and stride 1. Their (in-channels, out-channels) are (1, 6), (6, 16), and (16, 20) respectively, partially mimicking the LeNet architecture. The MaxPooling layers have kernel size 2. Other choices of these hyperparameters for the custom CNN were explored briefly too, but they gave no significant change on performance.

5.1 Quantitative Results

We summarize the performance of the representative models:

Trials	Train BLEU	Dev BLEU
1	0.6745	0.4697
2	0.7075	0.4635
3	0.7504	0.5236
4	0.8519	**0.5626**
5	**0.8638**	0.5564
6	0.7552	0.5386
8	0.6921	0.5082
9	0.6736	0.4602

Table 2: Models' Performance

In particular, the CNN with Transformer combination with 3 decoder layers fits the training data very well, and using a character embedding makes the model perform slight better on the development set. All models with a custom CNN encoder improved significantly upon the baseline models (models 1 and 2), but the models with ResNet or VGG as the encoder did not improve much upon the baseline models.

See Figure 5 for the training visualization of the best performing model 4. The figures are generated via `tensorboard` (so axis labels are not included, but it should be clear from the caption what the vertical axis is, and the horizontal axis is always the number of training steps.) Models are trained using PyTorch [8].

5.2 Qualitative Results and Analysis

We observe that for our best performing models (custom CNN with either characters or custom vocabulary, with either max or random sampling), the machine generated Latex code mostly compiles. The diagram however, can be incorrect. Here is an examples in the development set: We see that

$$\begin{array}{ccc} \mathcal{I}1 \otimes \mathcal{O}X\mathcal{F} & \xrightarrow[\text{id1}]{} & \mathcal{I}_2 \\ & & \downarrow{\scriptstyle \text{id}_1} \\ \mathcal{I}_2 \otimes \mathcal{O}_X\mathcal{F} & \xrightarrow{\text{id}_2} & \mathcal{I}_2 \end{array}$$

(a) Diagram generated by predicted code

$$\begin{array}{ccc} \mathcal{I}2 \otimes \mathcal{O}\mathcal{F} & \xrightarrow[c_2]{} & \mathcal{K}_2 \\ \downarrow & & \downarrow \\ \mathcal{I}_1 \otimes \mathcal{O}\mathcal{F} & \xrightarrow{c_1} & \mathcal{K}_1 \end{array}$$

(b) True diagram

in this example, the model recognized the general shape of the diagram, and got the nodes mostly correctly. However, the predicted $\mathcal{O}_X$ and $\mathcal{I}$ instead of the correct $\mathcal{O}$ and $\mathcal{K}$ shows the model is overfitting: The notation $\mathcal{O}_X$ is extremely common in algebraic geometry, and it appears a lot of times in the training set. On the other hand, the notation $\mathcal{K}$ is rare, and the model didn't manage to generalize and output the code for $\mathcal{K}$.

Another example is the following. The model made a mistake because the node X is very common

$$\begin{array}{ccc} X & \xleftarrow[a]{} & T \\ {\scriptstyle a}\downarrow & & \downarrow \\ X & \xleftarrow{b} & T' \end{array}$$

(a) Diagram generated by predicted code

$$\begin{array}{ccc} V & \xleftarrow[a]{} & T \\ {\scriptstyle h}\downarrow & & \downarrow \\ U & \xleftarrow{b} & T' \end{array}$$

(b) True diagram

in the training set, but V and U appear not as frequent. This again shows the model has a certain degree of overfitting.

Notably, the models with the more powerful encoders (VGG or ResNet) did not perform much better than the models with no encoder at all. A reason for this can be we did not have enough data to successfully train such deep networks as VGG or ResNet, so the performance of VGG and ResNet is not much different from a null encoder. On the other hand, using pretrained weights for VGG and ResNet is also problematic, because those weights were obtained by training on ImageNet, but the images of commutative diagrams used in our task are significantly different from most images on ImageNet: they are only black and white, and they are sparse since only the few pixels outlining arrows and letters carry information. Since our data is not from the same distribution (rather, far away) as where the weights were trained on, the pretrained weights did more harm than good. In contrast, our small hand-designed CNN worked much better, as the amount of data was suitable for training a network of this size.

In all models we trained, we observe some degree of overfitting, since they training BLEU score is considerably higher than the development BLEU score. We used weight decay and dropout to mitigate overfitting, but they ended up only decreasing training performance but not boosting development performance. We should also notice that the model that used random instead of maximum sample is a more regularized model, achieving the same level of development performance as the best model with a lower training score. Random sampling implicitly regularized the model by possibly preventing it from using the most common pattern (e.g. `Spec`) in training set whenever the model is unsure. Using a custom vocabulary speeds up training since it saves the time of learning those common patterns, but it results in a less regularized model, for the same reason as above. This agrees with the actual result that the model with custom vocabulary achieves highest training BLEU score but the one with only characters achieves highest development BLEU score.

6 Conclusion

We tackled the task of generating Latex code for a given commutative diagram. The best performing model used a CNN encoder and a transformer decoder with character embedding, and it achieves a high BLEU score on the development set and the training set. The generated codes for the development set are mostly compilable and the generated diagram generally has the correct shape, althought it might contain some incorrect labels and arrows. For feature work, we may try to regularize the model further, gather more data of diagrams and codes from mathematical areas other than algebraic geometry to diversity the dataset, attempting to build a model that can generalize well to unseen patterns.

7 Contribution and Note

This is project is done by the author along, without any external help or any collaborator. This project is shared with CS229. All material presented in this report should be considered as being done for this class, but the material other than the designing and training of models using a random sampling will be used also for CS229.

References

[1] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator, 06 2015.

[2] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention, 2015.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[4] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

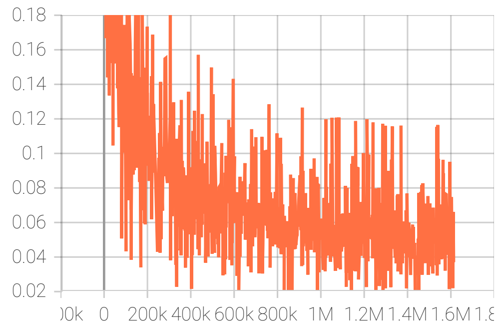
[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[6] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

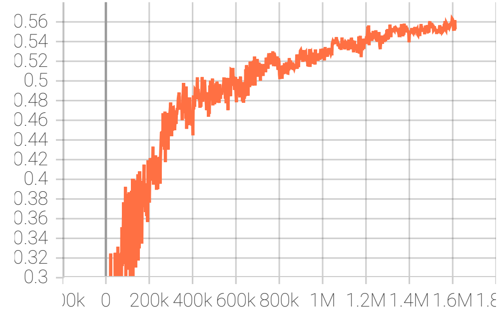
[7] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational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

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

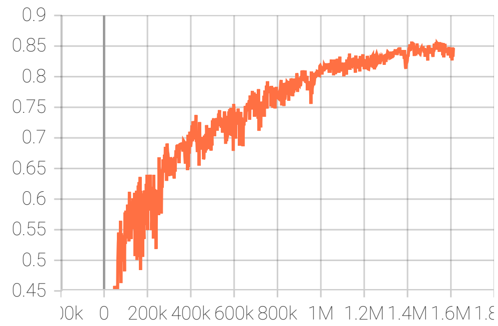
A Training Visualizations



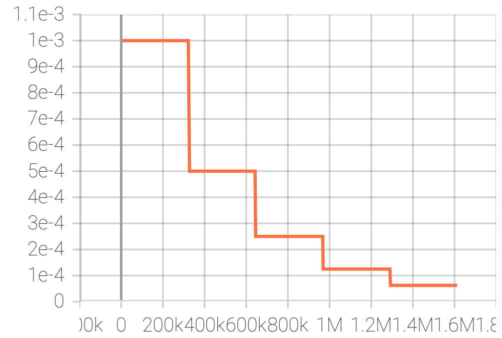
(a) Train Loss



(b) Dev BLEU Score

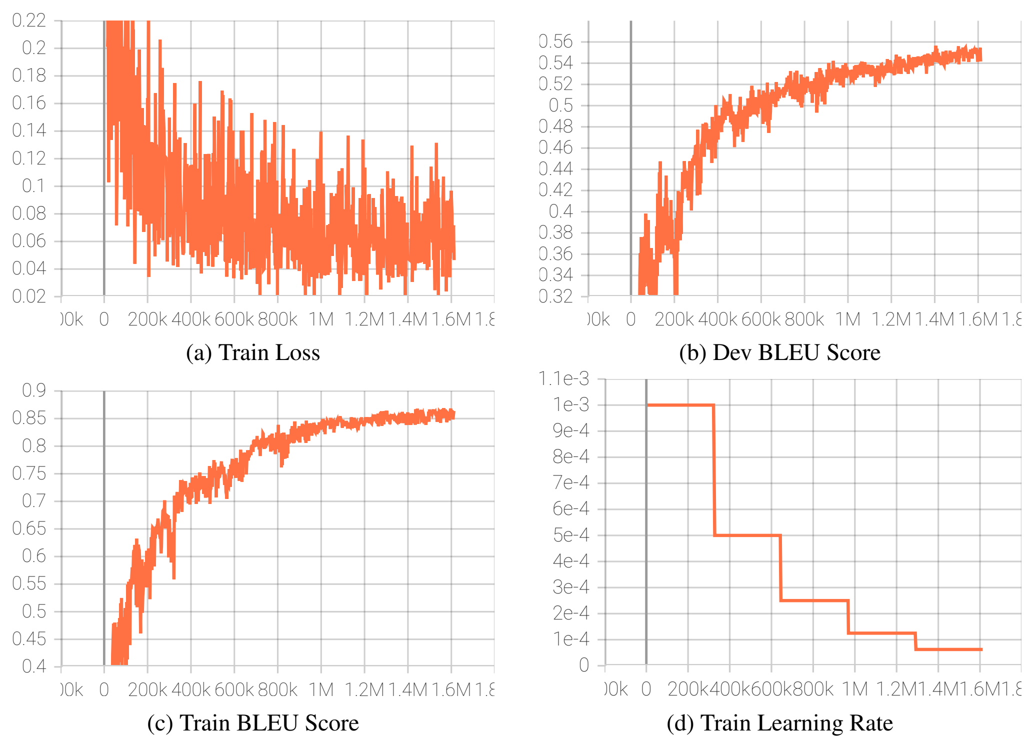


(c) Train BLEU Score



(d) Train Learning Rate

Figure 4: Best Performing Model Training Visualization



(a) Train Loss

(b) Dev BLEU Score

(c) Train BLEU Score

(d) Train Learning Rate

Figure 5: Model with Best Trainig BLEU Training Visualization