Generative Math Model for 1D Algebraic Equations

Gabriel de Sa
University of Texas at Arlington
Professor William Beksi
Introduction to Artificial Intelligence

Abstract

Report on the Fall 2021 Semester project for Artificial Intelligence under Professor William Beksi. Focuses on a fine-tuned Sequence to Sequence model with custom data from Math Dataset provided by the Dataset library. Training data consists of primarily one dimensional linear algebra questions posed as equations. The project consisted of editing the data set and fine tuning a base T5 model. The report goes over the existing technologies used, and their specific implementations in training the model. During training there were hardware and time limitations, however the model was able to predict some responses to the desired specifications, and with more training would have had a higher accuracy.

1. Introduction

The initial idea for the model began when working on unit conversions for a freshman level chemistry course. When tasked with learning a subject in math, most of the work comes from problem solving. Without being exposed to a variety of problems, most people won't really learn the material. However when trying to find additional resources online, it can be cumbersome or expensive. The basic idea for the model was to generate equations similar to the ones presented by the user. The initial idea is to have some hard-coded algorithm that tries to replicate the equation fed into it, but this quickly fails when the model needs to consider a wide variety of equations, with similar context. Basically the hard coded algorithm could work, but it would have a lot of edge cases with very little adaptability. So in order to generate the equations quickly and in a reliable way, I thought mimicking natural language would be the most intuitive way. My initial thought was to generate a transformer model that could iterate through the input string and attentively produce an output. This is the basis of the transformer model (Vaswani et al., 2017) , which iterates through an input, encodes it and generates an output, which is a product of its previous outputs. This way the model maintains context while generating the next words. This process makes it ideal for a problem where the equations can be very variable yet, are classifiable within similar subspace. For example one dimensional linear equations has at most one variable. However it can have any combination of operator or constant/coefficient values. The focus then begins on what kind of pre-trained model would be the most optimal to fine-tune to this specific problem. I reviewed many different transformers such as Bert and GPT-2. Each have their own benefits, Bert can classify very well and GPT-2 is great for generation. However the more I looked, the more a sequence to sequence model would be a better fit for the problem. A sequence to sequence (Seq2Seq) model works well for solving machine translation problems. I thought that the problem most matches a translation problem, but instead of translating between languages it translates between mathematical equations. Since the transformer package in python has a very limited number of pretrained seq2seq models, I decided to use the T5 Base model, which we will go into more detail in the literature review section. Next we will state the problem in a formal fashion, look at the solution that I derived and some experiments performed on the model. Finally we will look at look at results, and what could be improved.

2. Related Work and Literature Inspiration

We will review many of the foundations that create both T5 and the idea behind the generation model.

2.1 Attention is All You Need

Important literature for solving the generation problem begins with Attention is All You Need (Vaswani et al., 2017) which introduces the first transformer model. A synopsis of this model is that it stacked self-attention and point-wise, fully connected layers for both an encoder and a decoder. The transformer model does not contain any convolution or recurrence, it needs to including positional encodings to pass information about the relative and absolute position of the tokens in the sequence.

2.1.1 Tokenizers A very important component of training a transformer is the tokenizer. A transformer does not use text to generate its outputs. Instead it tokenizes the input and generates a numerical equivalent of the input. Each character has some equivalent value in the transformer vocabulary. Having tokens allows to use special tokens as deliminator for the start and end of the input/output. We can also use the token values to calculate for error.

The transformer model uses multi-headed attention layers and position-wise feed-forward networks to pass the attention information through the model's layers. Each layer in the encoder and decoder attention layers, the inputs move through the network from the previous decoder layer, with memory keys and values from the output of the encoder. The encoder also contains self-attention layers. Each position in the previous layer of the encoder can focus on all positions in the current layer. Self-attention layers in the decoder also allow position to transfer from one layer to the other. In the paper, more detail of how information is maintained and loss minimized. In general the Attention is All You Need paper set the standard for attention based models, and shifted the industry from Long-Short-Term Memory models to transformer models.

2.2 Analysing Mathematical Reasoning Abilities of Neural Models

David Saxton et al. 2019, another set of engineers members of Googles DeepMind division. Began working on the premise of humans needing to learn math from a symbolic standpoint and not a evidence standpoint. The majority of humans observe learn mathematics on the "basis of inferring, learning, and exploiting laws, axioms and symbol manipulation rules." Basically stating what I had above that in order to learn mathematics well, most people need to practice and learn "short-cuts." Their paper focuses on using Neural Networks to learn the symbolic representations of maths. In their research they created a Dataset of questions and responses for testing two seq2seq models. They tested the Attentional LS TM and Transformer model with their dataset. The majority of what I found from this paper was the Dataset, and the use of sequence to sequence models for fine-tuning my model. The dataset that the Saxton et al. released was made specifically for seq2seq models, which consisted of training data, as well as two different types of tests. Theses tests tested the types of questions that occurred in the training, and the other test contained questions that changed in difficulty to try and test the reasoning of the model, and its ability to learn more than what it was trained. The data was in the shape of question and answer as state above, where the answers were (at least for the one dimensional linear algebra dataset) were single valued real numbers. The premise of the paper was to list its contributions to the mathematical expression evaluation field of Artificial Intelligence and Machine Learning using symbolic processing in the form of transformer and natural language processing models. From the paper the structure of the questions were in the form Let X = ¡description¿, ¡question(x)¿ which basically meant that it defined x, and then asked a specific question to solve from the description. An example would be Solve 32x + 1 = 3 for X. This format fit exactly into the shape of the data I wanted to use to train the model. Since the dataset was originally created so that the answers wouldn't be very complex, Saxton et al. evaluated the model by seeing the output character for character with the expected answer. This would not work for the adaptation I made to their model, but it did set a standard for the results. From their results, the model that performed best was the Transformer model with the interpolation testing set generating the correct answer 76% of the time. While the Attentional LS TM with a LS TM encoder having a interpolation across modules of 57%. Seeing this result is what guided me towards looking into a pre-trained sequence to sequence transformer model. Since the dataset exists and was already tested on existing models, I decided that it would be best to continue with their highest performing model.

2.3 Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer

The paper by Colin Raffel et al. 2020 focused on transfer learning, which consists of a pre-trained model on a data-rich task that is then fine-tuned to focus on a more specific task. This is basically a model that tries to learn many things so that its weights are balanced in a way where learning a more specific task can be done less expensively. This makes sense as that a model that has been exposed to many things, might be able to reduce the cost of fine-tuning since theoretically one could narrow the model down with less data. Pre-training has shown the best results when done with unsupervised training on unlabelled data. This allows the model to be more general for specific "downstream" tasks. The paper goes into detail on how rapid progress in transfer learning makes it difficult to determine which algorithms are best for natural language processing, and it is an attempt to systematically study the different approaches used in the field. The premise to to treat each any text processing problem as a "text-to-text" problem. The model they used basically took any task and training the model to generate some amount of text relating to the task. Be it translation, summarizing or determining the correctness of a sentence. The idea was that their model was dub the "Text-To-Text Transfer Transformer" or T5. Their model was created to test the current state of the AI field, however in testing the field they created a model that was generic enough to understand and complete many tasks. This is what drew it to me, since it is pre-trained I could fine-tune the model to better fit my needs. T5 is an encoder-decoder transformer that is similar in structure to the original transformer model. Their model is trained with a large collection of unlabelled data for supervised learning. Raffel et al. utilize a cleaned version of the Common Crawl dataset, which is a dataset that contains 20 terabytes of public website data. Raffel et al. had a set of heuristics for cleaning up the data which can be found in their paper. The model was fine-tuned to have a certain structure to their questions. For example including "translate english to german" in the input, would generate the desired output. However they found that their text prefix was more of a hyper parameter and they did not do extensive testing on its impact on model performance. (I will address this later in section 5.) Their base model encoder and decoder modules consist of 12 blocks. As stated in the paper, it is similar in configuration to a BERT BASE model. The pretraining data consists of 35 billion tokens that are never repeated. Raffel et al. use an inverse square root learning rate schedule. Which is constant for some number of steps then exponentially decays. They learning rate used when fine-tuning was .001, with checkpoitns every 5000 steps. Some of the unsupervised results from the model had a varying results but the majority performed better with pre-training. Something of note is that the model generates the original text some of the time. As we will see later, generating the input text is common for the model, and is done frequently when under-trained.

3. Problem Statement

The use of natural language processing to try and approach solving mathematical equations is a broad and reaching topic. The purpose of this paper is to see if a model can retain the rules and symbolic mathematical representations necessary to generate equations from a set of known algebraic equations. In order to generate a math equation effectively, not only does the model need to understand the context in which the equation is placed under, but also the rules and structure of mathematics. A general purpose model could effectively solve this problem. In the following sections of this paper, we will look at the preliminary solution, results and future work.

3.1 Example Problem

An example problem the model should take is generating a custom equation given the input example equation. The format of the question would be

$$Solve\ 2x + 1 = 3\ for\ x. \tag{1}$$

which should return a similar equation. Since the time was limited for this project, the model will only focus on one-dimensional linear algebra equations like the one above. For example,

$$Solve\ 3 - 4x = 5\ for\ x. \tag{2}$$

The important part here is that the context of the equation exists, and that the we know the product is also a one dimensional linear algebra equation. With this in mind, we begin the steps taken to fine-tune and train the necessary models.

4. Model Implemenation

To begin the process of solving this generation problem, I had to look at different models and training sets. For a while I was not able to find the necessary dataset. So I spent quite some time creating an algorithm to generate my own data and supply it to the model. This data was of the shape

$$3x + 1 = 4 => 6 - 1x = 5 \tag{3}$$

this was a purely mathematical representation of the problem, with no written context. However with more research I found that generating my own data might lead to over-fitting the model, as well as inconsistencies from the data. Once I began working on the implementation of the model, I found the math dataset. After further researching the dataset I began the process of fixing and updating it to meet my needs.

4.1 Re-purposing the Dataset

As stated in 2.2 the maths dataset was initially created to test the mathematical reasoning of the transformer and similar attention models. Since the dataset was purpose built for question/answer models, I followed the steps highlighted in the paper to try and reproduce its accuracy with my custom recreation of the dataset. In order to meet my needs, I took the 1D Linear Algebra Subset of the Dataset, and mapped all the answers to a randomized version of the questions. This process converts the answer from the dataset, which were of the form {Question: ”Solve 2x + 1 = 3 for x.”, Answer: ”1”} to {Question: ”Solve 2x + 1 = 3 for x.”, Answer: ”Solve 1 + 3x = 5 for x.”}. This process required making two copies of the dataset, which was 1.9 Million Examples. Then shuffling one of the copies, and creating a mapping function to adjust the main copies answer feature.

4.1.1 The Dataset Library An import note here is the use of the datasets library available in python. This library contains the necessary tools to download and process the datasets available on Hugging Face. Whenever you download a new dataset, it will store it on your local system. This allows for quick access on the next initialization of the object. The actual dataset is a object is a dictionary called DatasetDict, which for the dataset I used had the keys Train and Test. This comes from the dataset that was created by Saxton et al. Where the training data is 1.9 million examples, and the testing data contains interpolation and extrapolation tests. The testing dataset had 10,000 examples. Each key in the dataset dictionary is itself a Dataset object. This dataset object contains the features array which shows the columns, and the number of examples. Each dataset is an array, where you can access the example with normal indexing. In our work we segment the data into training and testing batches. I will not use all 1.9 million examples to train, instead I chose a smaller batch of 10,000 training examples.

4.2 Tokenizing the Data

In order to pass the data through the model we have to tokenize and label the question/answer. For this we have a network that tokenizes the data itself. This algorithm itself splits and separates the input text into numbers the model was trained on. We can use the tokenizer.from_pretrained function to use the same tokenizer that our model used for pre-training. This will split the input in text that is understood by the model. The dataset used contains some tokens for the beginning and end of the output/input string. This is a b’ and ’//n token. Since the T5 model is not familiar with these tokens, we had to add them to the tokenizer as special tokens. These special tokens are then also added to the model so that it can recognize them during training.

4.2.1 The Importance of Tokenization Tokens are extremely important for training transformer models since making mathematical calculations using numbers is a better way to calculate error and loss during training. When inputs are mapped to a certain numerical representation, the model can also create consistent outputs. In fact something interesting about these models is that even though the outputs are decoded into text for human interpretation the model itself never sees anything but numbers. Meaning that the process of even translating English to French, is actually translating a certain combination of numbers to another for the model. This means that the model needs to be able to interpret the nuances of human language using only numerical values. Which is yet another benefit for the model, because depending on the sequence or context behind the token, it might assign different values to the same word. This allows the model to truly have attention and context with different tasks. Tokenization is different for each model, so it is important to use the same tokenizer as the model you are using.

Once the tokens are added to the model, we can then actively tokenize the model by mapping each dataset example and passing it through a function, that also generates the attention mask, the input ids and the labels.

1. The input ids are important because they are the tokenized value of the input. The model usually bases its training size off the size of the first input id. So you can either apply padding equal to the largest training input, or apply a max size padding.
2. The attention mask is generated with the input ids and uses a matrix of input by input length. It is usually a triangle matrix that has either true or false values. The attention mask basically gets convoluted over the inputs. Each time the input gets more and more attention.

$$\begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{matrix}$$

The matrix as seen to the above allows you to iterate through the data with increasing importance of the input id.
3. The function also produces the labels, which is the tokenized equivalent of the expected output. This is needed in order to use supervised learning to train the model. Without the labels the model doesn't know how much to correct and will continue to produce either random noise, or not work at all. Since the labels are in the same tokenized form as the rest of the model, computing the error is simply of a matter of computing some error function of the model training outputs and expected answers. This will allow us to effectively fine tune the model.

4.3 Model Initialization

The model that I choose to use during the first training was the T5 model. This is because as part of the dataset creation was done with a sequence to sequence model. The models available on hugging face for pre-trained sequence to sequence models were T5. Since T5 is a generic pre-trained model (at least the base model), I thought it would be the easiest to fine-tune for the problem. The T5 model in general comes with many different forms of pre-trained tasks, which include translating and summarizing. Since these tasks are pre-trained, and can generate new inputs. Once the model was initialized I then had to adjust its vocabulary. The T5 model has a vocabulary size of 32100. With the additional tokens added, the model used for training had a vocab size of 32102. Once the model is initialized using the Trainer package in Python, we can begin the process of fine-tuning the model.

4.3.1 Trainer The Trainer class comes from the Transformer package in python. You can import it as you would any other class from that package. The trainer class simplifies the training process. You can import a TrainingArguments class that contains all the information you need to train the model, this includes the learning rate, the number of epochs, the batch size and many more hyper parameters. The TrainingArguments class needs at least the output directory for the training results. You then can pass these arguments into the trainer upon initialization. The trainer also takes the model, the arguments, the training dataset and the testing dataset. Our Training and Testing dataset were both 10,000 examples. We can then use the trainer to actively train the model.

4.4 Training the Model

Once the trainer is initialized we can begin the process of training the model. The trainer takes care of all the initialization and training processes. You simply need to type trainer.train() and the model itself is updated. This takes quite some time and a lot of computational power. Using pytorch we are able to place the load on the Nvidia GPU using CUDA. CUDA is a machine learning interface for Nvidia GPUs that expedite training speeds and decrease resources needed to train Neural Models. I trained the model with a Batch size of 8 for 3 epochs. Meaning that the 10,000 training examples we had passed through the model 30,000 times. This took 4 hours on my hardware. The loss during this section of training decreased with each epoch. However the loss decreased too much and the model no longer produced any meaningful results. This meant that we had to start over, up until this point I had not yet realized that I had to add the tokens to the model, I did this and the results turned out better than the first time. This means that the model needed the tokens in order to make meaningful outputs. However the loss was high.

5. Results

Here we will go over the general results found while training the model. This includes all inaccurate and accurate results.

5.1 Initial Training

Since the T5 model comes pre-trained we can take advantage of transfer learning to try and fit the model to our problem. However initially the inputs were incorrect and the model did not learn effectively. In this batch we trained over 30,000 examples, saving a model every 5000 steps. Given the training evidence the model seemed to be performing extremely well, but once testing began I quickly realized that the model was just giving NaN values for the outputs. This means the model was either producing values too small or too large. Either way since the answers were NaN I was not able to do any further testing. I had to start from scratch. The overall training loss was .0676 however, this is an inaccurate number since the training loss for the majority of the epochs was zero. Meaning the model quickly over fit the data, and couldn't adjust its weights accordingly.

5.1.1 Changing Hyper-parameters and Tokens This is where I found that I needed to change the tokens that the model accepts, as well as include a performance metric for the trainer. In order to get the best performance out of the trainer I had to initialize the training batch size to be one. This is only because of hardware restrictions with both my hardware and Google Colaboratory.

5.2 Final Training

For the final training of the model, I created a sample of 1000 training examples. I pushed this through the trainer and the loss results are shown below.

Epoch	Training Loss	Validation Loss
1	.08740	.09635
2	.07860	.09361
3	.07540	.09324

The overall training loss for this iteration was 0.0898, this is a very good training loss. However this time the model actually produces valuable results. Since the loss decreases throughout the epochs, the model was fitting the data better with each iteration through the training set. An issue might be that the training set only contained 1000 examples, while the testing set contained 10,000 examples. However this training method was more accurate with a smaller training set than the larger one. Also having the hyper-parameters adjusted increased the models performance. The training and validation loss decrease with every epoch, by a small percentage which means the difference between the epochs did not improve significantly. As we will see in the experiments section that this simply could mean that the model was giving similar answers to back based of the input.

6. Experiments

In order to test the accuracy of the model, I created an interface to test the model. Simply input the equation you'd like the model to generate and it returns the result. This is important because I can use non determinism to generate an equation from the top of my head. This allows me to generate new results that most likely wasn't seen in the training or validation data.

6.1 Random Samples

To randomly test the model, I generated some random examples and passed them through the model.

1. Solve 3*x + 1 = 4 for x.
2. Solve 1 + x = 4 for x.
3. Solve 55*100 + x = 20 for x.
4. Solve y + 1 = 5 for y.
5. Solve 100*z + 3*z = z for z.

Responses Were:

1. b'Solve -0 = -0*w - 0
2. -x = -x + x for x. -x.
3. Solve 0 = -0*0 for 0.
4. Solve -26 = -27*w + 0 for w.
5. Solve -26*w = -27*w + 0 for w

In general I would say that the model generated a cheat to solving the problem. If it generates a new similar response every time it gets called, it will cheat the loss algorithm. Since the model fits random data, there isn't really a way to guarantee the accuracy of the response. This means that the model might start producing the same response with every similar equation. This means x + 3 = 4 might generate the same response as y + 4 = 5 since they are of the same form. However the results are uplifting and show potential with more fine-tuning of the hyper-parameters. Another important note is that although the model does generate valid equations, the actual value of the equations generated is limited. Since a lot of the answers have zeros, or the same variable throughout it doesn't seem that the model learned any of the important mathematical rules. These include division by zero, and having at least one variable.

7. Conclusions

In general I am happy with the progress made with this model. Although it doesn't understand all the rules of mathematics, it does still generate new equations that are not one for one copies from the input. (This has improved since the previous demonstration due to more fine tuning of the model.) I think that the general performance of the model indicates that there is a lot of room in which the model can learn and adapt to even larger subsets of the math_dataset. I think that the fact that the model did generate the same equations, means that the function that was being used to calculate loss was not effective enough. Or that the answers in the dataset did not account for the probability of the model generating the same output consistently.

7.1 Future Work

Some future work that I could see for this model would be increasing the size of the problem space, meaning we could add more than just one dimensional linear algebra equations. We could also fine tune either the loss function itself or the hyper-parameters to try and maximize the generation of new and random equations each time. Trying the models in different stages of training might also lead to different results. I could also test the randomness of the model by seeing the difference between similar equations and work

7.1.1 Evaluation using Trainer In order to evaluate the model effectively, I could use the evaluation feature with the trainer class. This allows us to use a compute metric to evaluate the accuracy of our model. It takes the training, and evaluation data and computes the metric on every epoch. This metrics function computes the logtis and the labels for the data, which is then passed into argmax and returned to the trainer. This enables the trainer to see the accuracy of the model and generation some evaluation value. This evaluation can be done

7.2 Errors and Problems during Training

As stated previously, the tokens played a major role in the accuracy of the model. Once the tokenizer and the model were adapted to include the new tokens the performance greatly increased. I think that the model works decently well given the time constraints of training and resources. Another issue is that batch sizes on both Google Colab and my local computers couldn't handle anything larger than 4 examples. This means that the model adjusted itself with every example. Which can lead to loss calculation errors. In the future if I had a stronger computer I would try to match the fine-tuning specification used in Raffel et al.

7.3 Final Thoughts

The fine-tuned T5 model generates responses to the equations, although sometimes nonsense. Even though it isn't the most accurate I think we made great progress and pushed the field forward. I went through plenty of papers and research and didn't find anything on generating mathematically correct equations via a transformer model. A link to a Github Gist of the Google Colab can be found https://colab.research.google.com/gist/Gabriel-Sa/32d21f8e786d863c58bad16ac9a436da/gmm.ipynb. This will show all the implementation details of the project.

8. References

1. Saxton, D., Grefenstette, E., Hill, F., Kohli, P. (2019). Analysing mathematical reasoning abilities of neural models. arXiv preprint arXiv:1904.01557.

2. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., ... Liu, P. J. (2019). Exploring the limits of transfer learning with a unified text-to-text transformer. arXiv preprint arXiv:1910.10683.

3. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. In Advances in neural information processing systems (pp. 5998-6008).

4. Math_dataset · datasets at hugging face. math_dataset · Datasets at Hugging Face. (n.d.). Retrieved December 11, 2021, from https://huggingface.co/datasets/math_dataset.