CS 352 - Machine Learning
Final Project - Lab Report
December 23rd, 2018
Malcolm Gilbert and Weronika Nguyen

Different approach for Image to LateX via Neural Nets

Introduction:

LaTeX is a markup language widely used in mathematics and science because of its ability to display complex mathematical expressions. However, these mathematical expressions are much more difficult to write in LaTeX than via handwriting. Thus, having the ability to quickly convert from an image of a mathematical formula to typeset LaTex would be incredibly helpful. The first step of the problem is the conversion of computer-generated images of mathematical expressions to typeset LaTeX, the problem which we will be tackling here. As there are multiple people who have attempted this problem, we will be building upon their previous work.

The previous work in question is Avinash More's architecture presented in his *Image to LateX via Neural Network*[1] paper. From the outset, using his in depth paper to build and then expand upon his models seemed like a straightforward task. He used TensorFlow, but since we were more experienced in TFLearn, we decided to implement his architecture in TFLearn for simplification. Where he started with, and where we started off with, was creating a very simplified model where we simply predicted which of four characters the generated equations started with. We will go into more depth about how we generated these images and and the architecture we used.

Then, our next step would be to build multiple neural networks to recognize each separate character training for each LateX formula. Then we will try to combine the independent character predictions to form a general output for the whole equation. Thus this turns the image recognition problem into one large classification problem.

[1]More, A. 2018. *Image to LateX via Neural Network*. San Jose State University SJSU ScholarWorks. https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?referer=https://www.google.com/&httpsredir=1&article=1599&context=etd_projects

Methods:

In order to simplify the problem, instead of using the images and formulas provided in HW6, we generated our own. We used the data generation file from More's github, and added a few changes to suit our needs, namely formula padding and saved location, as well as updating it to python3. We generated four types of equations: fraction, integral, summation, and square root. Each formula would start with, for example, "\frac" for the fraction types, or "\sum" for the summation types. Each class of equations would have the same base components, but the variables and numbers used would be randomly generated. Thus, our first problem would be a classification problem, in essence predicting the first character. The target would be a vector of length four, where each place would correspond to one of the four types of equations.

Once we managed to replicate More's first step to implementing image to LateX neural nets model, we tried to solve a slightly more difficult problem. In particular, we aimed to predict an entire simple equation (using same data from the first problem) by building multiple models that would be trained on each character position separately. We used the same architecture as for the first character classification program but changed the number of units in the final layer to the number of unique characters used in all formulas. For this purpose, we turned our training and testing data into one hot vectors of dimension <max_formula_length> x <num_unique_chars>. Since we restricted our equation length to 16, we initially tried to create an array of 16 different models. We were planning to train and print the predictions for those models by using multiple for-loops. Unfortunately, we did not succeed using this approach as it turned out to be the case that we cannot train multiple models in a single python file. Given the limited time we had to complete our research, using the same file, we created and trained models for different positions of an equation consecutively by tuning in the training and testing data to the specific index in a formula and reported the performance of each model separately.

We reused the framework of the code from HW6, and updated the image processing and neural network model that we used. Initially, we replicated the model architecture from More's paper but, to our surprise, it performed very poorly and seemed to produce exactly same prediction for different testing images. Therefore, the model that we used was based off a tutorial we found for MNIST on the tflearn website[2]. Thus we used two convolutional, max pooling, and local response normalization layers. The first convolutional layer had 32 filters,

[2] https://github.com/tflearn/tflearn/blob/master/examples/images/convnet_mnist.py

each of size 3, and the second convolutional layer had 64 filters, each of size 3, The max pooling layer each had a kernel size of 2. Then we flattened the output of the last convolutional layer and fed it through three fully connected layers, with a dropout with probability of 0.8 in between the layers. The first, second, and third layer had 128, 256, and 4 (for the first problem) or <num_unique_tokens> (for the second problem) units respectively, with the first two layers had tanh activation and the final layer having softmax activation. Our final layer was a regression layer using the adam optimizer, a categorical cross entropy loss function, and a learning rate of 0.01.

To verify the model worked, we created a sample problem of the same size and shape as the problem we attempted to solve. In this sample problem, the images were 2d arrays of a single value, with the target being a binary vector where all the values were 0 except at the location of the single value, which was 1.

Results:

Our model for classifying the first character started to work very well after we switched from More's architecture to the MNIST one with local response normalization layers. It appears that these layers played a significant role in increasing the accuracy of the model since both More's and the MNIST model that we received in class (as part of the tflearn examples) did not perform as well on the problem as this one with local response normalization layers. As advised in More's paper, we also added training-data shuffling to make our model converge faster. We ran and trained our new implementation 5 times separately, each time for 10 epochs and batch size of 25; using training data of size 900 and validation data of 100 examples. The model consistently returned accuracy of above 99% on training data and predicted all the testing data (of size 200) perfectly - it yielded 100% accuracy.

As for the full formula prediction problem, after consecutively running models that were trained on a specific position, we received the following results:

position	training accuracy (%)	validation accuracy (%)	testing accuracy (%)
0	99.41	100	100
1	99.1	100	100
2	51.63	50	45
3	26.32	18	31.5
4	95	100	100
5	73.96	75	74
6	96.7	100	100
7	74.79	74	79
8	98.21	100	100
9	72.78	78	78.5
10	99.98	100	100
11	76.69	81	84.5
12	98.56	100	100
13	77.61	75	80
14	74.41	85	80.5
15	98.7	100	100
Average	**82.115625**	**83.5**	**84.5625**

Table 1. Accuracy report for a single run of separate character training.

Once we received separate results from running 16 models for each position (with the same image data but only with a part of the full label/targets), we took an average of all the training and testing accuracy and got about 82,12%, 83.5% and 84.56% on the training, validation and testing examples respectively. Although this approach is not very precise in returning the accuracy for prediction of each formula, it gives us a rough approximation of the percent correct. It appears that the model is able to produce more correct results for particular character places of a formula - it performs exceptionally well predicting the first, second, ninth, eleventh, thirteenth and the last token. On the other hand, it seems to have difficulty recognizing various variables and operators from the the third and fourth index, most likely due to insufficient data amount. Since we got 90% on predicting last four character positions, it does not seem to be the case that the deeper characters are harder to train like in More's model.

Tuning in the parameters:

- **Learning Rate:** Changing the learning rate for certain positions in according to More's suggestions (p. 46, Table 4: Character Position and the learning rate model) did not

bring any difference to the accuracy but slightly shortened the time required for the model to converge.

- **Feature map size**: By trying out different feature map sizes in our model for training the fourth position we got the following relation:

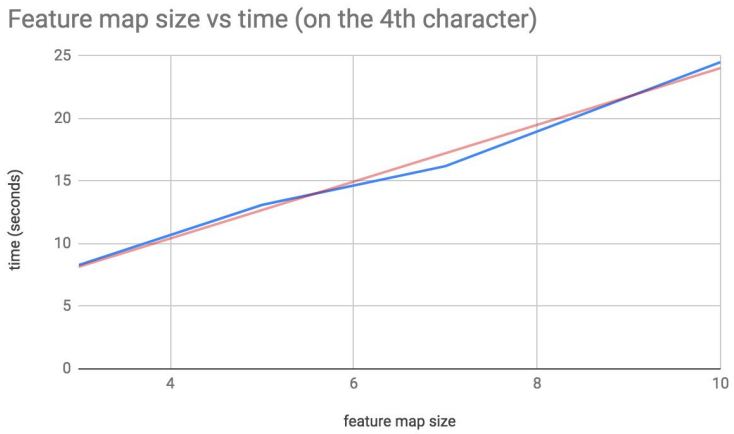

Image 1. Feature map size vs time taken (seconds) on the 4th position.

In Image 1. the blue line connects the results we got from changing the feature map size vs. time and the red line is a linear trendline. It appears that by increasing the feature size we are linearly increasing the runtime to converge. Changing the feature map size did not affect the accuracy of the model.

- **Batch size:** Below is a relation we received after changing the batch size for the model training the 4th position of a formula.

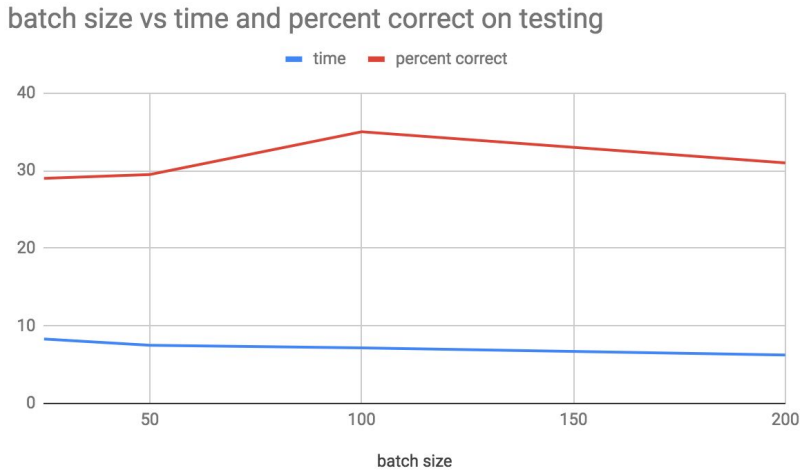

Image 2. Relation of batch size vs time and accuracy on testing.

We can see that the bigger the batch size, the faster the model converges. However, the converging runtime is not as important as the training accuracy level. It seems that the model works the best for batch size of 100 since it yields the best accuracy on testing data of 35% (for the fourth character).

Conclusion:

In our first problem of classifying the first token, we were successful in creating a model that would correctly classify the images according to their equation type. It appears that we outperformed More's model by using different architecture. We received accuracy of 99% whereas he reports 95% on training data and had to use 5000 images for it to converge.

In our second problem, we did not do as well as More, who got 90% on validation examples for each character and cost of less than 0.5. Nonetheless, we performed rather well on a single run with an average accuracy of 82,12%, 83.5% and 84.56% on the training, validation and testing data respectively. In addition, it seems that the approach of having multiple models yields better results than training the whole formula at once as our model from HW#6 could not make any valid predictions.

In the future, to improve our model we will test the relation of all the parameters for each index in the formula to find which variables yield the best predictions for which position. What is more, we will implement the Levenshtein distance metric method to better evaluate the difference between two sequences (prediction and result). We will also try to build a model for predicting much more complicated LateX equations.

Bibliography:

More, A. 2018. *Image to LateX via Neural Network*. San Jose State University SJSU ScholarWorks. https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?referer=https://www.google.com/&httpsredir=1&article1599&context=etd_projects