COMP 550 Research Project
Code Generation

Youngsun Jin
McGill University
youngsun.jin@mail.mcgill.ca
260616938

Allan Wang
McGill University
allan.wang@mail.mcgill.ca
260667681

December 5, 2018

Abstract

When learning new programming languages, it is often easy to understand the syntax and learn about idiomatic executions through examples. Particularly with different paradigms, how we write code in one language may not be one to one with code in another language. Most languages provide examples through a list of common executions. If we can instead provide an example matching simple instructions in English, we can help refine the amount of code a new developer has to go through, and assist in creating an infinite set of examples tailored to each user. In this work, we have developed a prototype system which is able to process natural language instructions related to list creation with optional conditions, which is then translated into executable code with programming language provided by the user.

I. Introduction

Learning the syntax of a programming language can be frustrating and challenging. Often, programmers need to know various programming languages that can greatly vary in terms of control structures, syntax, etc. One way to efficiently learn new programming languages is to go through a set of common examples. For example, it can vary from using simple conditional statements such as *if and else* to iteration structure using *for* or *while* loop. Indeed, almost every programming language supports these types of operations, but it can be often frustrating to ingrain subtle syntactical differences between programming languages, even for programmers with proficient experience. However, a natural language instruction is something that we can understand without much effort. Hence, we have built a simple prototype system which is able to translate a simple type of natural language instructions into executable code with programming language specified by the user. In this work, we focus on creation of a list of numbers with some optional conditions. For example, the instruction 'create an even list from 0 to 100' would generate all numbers between 0 and 100 that are even. Due to feasibility, we only allow a single instruction at a time, and we do not allow mutation. We explore methods of parsing natural language instruction, namely Context Free Grammar (CFG) and Combinatory Categorial Grammar (CCG). Then lastly, we evaluate our final model to systematically test whether or not it is able to parse natural language instruction with words that are not keywords.

II. Feature Design

For preprocessing we used the scikit-image [2] library to perform several filters and calculations to find the biggest foreground 'blob' in the image, to be able to reduce the problem down to digit classification. This preprocessed data was identified with a much higher accuracy than the raw data. Using scikit-image, we first applied filters like *sobel* and *watershed* to detect edges. We then applied *otsu-thresholding* to create separate regions between pixels with values greater than 250 and those smaller than 250 (since all digits in the data-set have white colors). This removed the noisy background from all the images. Afterwards we applied built-in region calculation functions from the library to detect the separated regions. Once all the regions were detected, we first attempted to find the smallest bounding rectangle for each region. We then sorted and picked the largest of these rectangles, thus giving us the largest image. This however did not yield great results as our preprocessed was showing the incorrect data. For example a '1' might have a smaller rectangular area due to it's shape than a

much smaller '5' so our models would only see the '5'. Instead we opted to use the smallest bounding squares for each region, and select the largest of the squares. Our preprocessing still failed to perform perfectly as it was unable to separate digits that were connected to each other in the raw data. They were instead considered one region by the library. However, the occurrences of such instances is minimal and hence training a model with the correct image for some digit for the majority of times that it sees it and a partially incorrect image the other times should not affect the model's learning of the class by a significant amount; yet it is definitely one area where our experiments could have be improved. One attempt to solve this included eroding the pixels slightly, but this created more problems than it solved.

An alternative approach considered for preprocessing was to simply clear out the background noise and leave out all digits in the image. This was a middle ground between fully calculating the image and the raw image method. We assumed that if the Convolutional Neural Network was deep enough the gradient descent would learn the features of the digits with enough iterations. However, this still did not yield results as successful as what we were able to achieve with the fully preprocessed data and hence the method was discarded in favor of our original preprocessing approach.

Additionally we performed Principal Component Analysis (PCA) on the preprocessed data before running kNN to reduce dimensionality so that calculations would be less redundant as well as faster. PCA wasn't applied before the other models as they shined with more dimensions and more information.

III. Algorithms

i. Support Vector Machines

Support Vector Machines (SVM) [7] are a way to separate the data with a boundary. When working with a linear kernel, with k classes, k different classifiers are trained by constructing an optimally separating hyper plane to separate one class from the k-1 others. For a given test case, the classifier which gives the largest value(i.e. largest probability that it belongs to that class as opposed to another class) is chosen as the class. In many cases, the data is not linearly separable, so we may experiment with the kernels. The data is transformed to a new space, and the data is separated with a linear boundary in the new space. In the original space this is a non-linear boundary. For Linear SVMs we tweak a parameter C, called the cost parameter, which roughly correlates to how much mis-classification happens around the boundary.

ii. Neural Networks

Neural Networks (NN) [3], also called *Multi-Level Perceptrons* are classifiers inspired by the biological neural networks of a brain. The core idea of neural networks is for a group of neurons to take an input vector and based on the values, produce outputs for the next layer. This mimics how neurons in a brain fire with an intensity, and potentially cause other neurons to fire. With proper training a Neueral Network can approximate any continuous function arbitrarily well. The two major processes to train a Neural Network are called *feed-forward* and *back-propagation*. Feed-forward is the process of a neuron taking a input vector, applying a weight vector to it, and applying an activation function to it, and outputting the value to the next layer. To learn and improve the weights, which are initially random, back-propagation is performed. The chain rule is used to calculate the gradient of error with respect to the given input in a reversed manner, layer by layer, starting from the output layer to the input layer and updating the weights accordingly.

iii. Convolutional Neural Networks

Convolutional Neural Networks (CNN), as the name suggests, are an extension of Neural Networks. They are specifically advantageous in the field of image classification as they take advantage of the image structure. Regular networks and other classic machine learning classifiers take column vector inputs while CNNs take 2-D or higher dimensional matrices. Convolutional layers scans the output of the previous layer in small windows(kernel), and produce an output value for each window scanned. Each scan is a dot product of the weights of the output neuron and the values of the region of the previous layer scanned. This is particularly useful in image classification due to local shapes in regions of images providing significant meaning

about what the image.

iv. k-Nearest Neighbors

The k-Nearest Neighbor[7] approach is a technique which does no actual learning. Instead it compares all our test data with our training data by means of a metric, and using majority voting on the k closest ones, selects the most likely class. The value of k and the metric are both hyper parameters. It becomes very prohibitive as the only way to classify each test example is to compare it with every single example of the training data.

v. Random Forests

Random forests are an extension of decision trees, where we simultaneously train multiple trees. Each tree is trained with a bootstrap of the original data and the best split of a random subset of features is used at each node. The final result is determined either by taking the majority vote or averaging the probabilistic predictions. While the randomness of feature selection increases the variance of the model, averaging effectively reduces such variance. The tree is allowed to run until maximum depth without pruning and the model does not over-fit with increasing number of trees.[6].

IV. Methodology

During the training of all classifiers except the Neural Networks, we trained hyper parameters with 3-Fold Cross validation, provided by GridSearchCV from the Sklearn library, and set aside 10% of the data as test data. For the Neural Networks we set aside 10% for validation to select the best models, and then tested the best ones on the Kaggle test set.

i. Support Vector Machines

For training the SVM classifiers, the data was resized to 28x28 pixels instead of the original 64x64. This was done as with higher values for the cost parameter C, the computations became more and more expensive. For linear SVMs we auditioned a wide range of values from 10^{-3} to 10^{2}.

ii. Neural Networks

Our implementation of Neural Networks was inspired by a book written by Michael Nielsen [5]. We first implemented the base classifier which supports using any number of layers or neurons per layer without modifications to the code. Although a bit more challenging to implement correctly, this method saves a lot of time in terms of tuning to not have to deal with the scripted implementation of the back-propagation algorithm for each additional layer added. For Hyper-parameters, we started the network with one simple layer and 30 hidden neurons in that layer. As expected, this model did not perform well at all. We varied our learning rate with 150 hidden neurons and achieved satisfactory results with a learning rate of 0.1. We believe that the base neural network classifier has the potential to get near perfect accuracy given enough tuning, however due to the restrictions of the project guidelines we were limited to running the training on CPU using basic implementations, which quickly becomes exponentially more computationally, and due to this it was not feasible to run numerous iterations for tuning this classifier given we could do better using CNN's and the Nvidia 1070 GPU that we had.

iii. Convolutional Neural Networks

We have used the Keras [8] library for creating our CNN models. The main tuning for CNN's involve trial and error on the network structure which include number of layers, layers' kernel size and strides, and dropout percentage. Starting with a single Convolutional layer and a single fully connected layer, we started by adding more layers up to the point where performance was getting bottle-necked by something else besides the depth of the network. To further improve performance, for each candidate CNN model we tuned the value on the dropout layer (which prevents over-fitting by dropping a percentage of the examples on each epoch) from 20% dropout to 70% dropout. Most of these models achieve very high training accuracy with enough epochs and therefore it is necessary to insert methods for reducing the over-fitting. We also used L1 and L2 regularization on the dense layer for our best model using 5 lambda values in a logarithmic space from 0.1 to 0.001 however this drastically reduced performance and was

discarded from our final model. For activation functions, we tuned on using the *ReLU* and *Leaky ReLU* variant. Both activation functions yield similar results however in most our submissions we picked *Leaky ReLU* as it is safer in preventing dead neurons in the network. We have also tuned on different optimizers including Stochastic Gradient Descent (SGD), Adams, and Nadam. SGD with a momentum value of 0.9 and Nesterov momentum applied yielded the best results. The final structure of our network is shown in figure 1 below.

We experimented with data augmentation using *ImageDataGenerator* provided by Keras [8]. The idea of data augmentation is to apply image operation on a base data-set so as to obtain a larger set of data. We tried using both MNIST and the training data provided as the base data-set. We also tried various operations such as rotation, transformation, and zoom. The data augmentation does not improve our accuracy. It may be because when we are transforming the image, it becomes too different from the actual data-set.

iv. k-Nearest Neighbors

For k-NN, there were two steps used to achieve good and efficient results. The first was to process the data with PCA as it would allow us to reduce the dimensionality of the problem, separate variance, and make computations much faster. Due to the size of the images, if k-NN was to run on the raw images it would take many more hours to run far fewer splits with GridSearch, and results would generally not be as accurate. The metrics that were auditioned were minkowski, hamming, manhattan, canberra, braycurtis, euclidean, and chebyshev metrics. The k values auditioned were 5, 6, 7, 10, 15, 25, 100 after it was observed that lower values performed better with this data set.

v. Random Forests

We used sklearn to implement random forest. We set 0 as the seed for the random number generator. Doing so allows us to get more consistent results. We mainly consider two hyper parameters: n_estimators, namely the number of trees in the forest, and max_features, which is the number of features to consider when looking for the best split. Sklearn's GridSearchCV with 3-fold cross validation over the entire training data is used to exhaustively search over the parameters. For computation, as with SVMs, we reduce the image size to 28x28 from 64x64. Doing so allows us to do a more intensive gridsearch in a much shorter period of time with almost, if not all of the accuracy.

V. Results

Please refer to the Appendix section for more figures.

i. Support Vector Machines

After performing a GridSearch on the values for the Linear SVC, we found the optimal value for the cost parameter C to be 0.01 it gave a test accuracy of 0.7712. This data is likely not linear separable.

ii. Neural Networks

Our base NN model was prohibitive to run as the application was single-threaded and unable to utilize the GPU. Running the model with more than one layer would take several minutes per epoch, but we performed several tests with one hidden layer having 150 neurons while varying the learning rate from 0.0005 to 1. These are the test validation accuracies we found.

LR	.0005	.001	.01	.1	1
	.694	.7358	.8692	.919	.9006

We can try with more neurons and more layers but we will move onto the Convolutional Neural Networks and libraries which allow us to delegate tasks to the GPU.

iii. Convolutional Neural Networks

Our "deep" CNN model (MULNet: Figure 9) had the best performance of all with a validation score of 96.15% and submission test score of 95.03%. In figure 13 and figure 14 we've shown the validation and training performance on last epochs for all CNN models. With dropout percentage tuning and comparison between Leaky ReLU and ReLU, we achieved the results shown in figure 3 below.

In figure 2 we see that the "Deep" (MULNet) model performs much better than any of the other

models, and figure 3 shows Leaky Relu activations with 70% dropout consistently performs better than the rest of the configurations on the MULNet model. Figure 15 illustrates the accuracy performance of the Deep (MULNet) model using different optimizers. We can see that the SGD Optimizer with Momentum value (0.9) and Nesterov momentum applied. From the above tuning we conclude that the "Deep" (MULNet) Model, with Leaky Relu Activation, SGD Optimizer with 0.9 Nesterov Momentum, and 70% Dropout has had the best performance of all our models in this project.

iv. K-Nearest Neighbors

First we performed PCA on the data to make K-NN run faster and more accurately.

We found that converting our data to 50 components performed the best and allowed us to perform the most thorough sweep of hyper parameters while also improving our accuracy. After performing a thorough grid search on the data we find that the braycurtis metric,

$$\frac{\sum |x-y|}{\sum |x| + \sum |y|}$$

performs the best with a k value of 5.

When testing the best values with a test split, we have an accuracy of 92.51%.

v. Random Forests

Without any tuning we obtain 16.64% for the model. We noted that the validation accuracy increases with the number of trees. For max_features, sklearn defaults to sqrt(features) but we also tried $\log_2$(features) as well as a range from 100 to 500 during our gridsearch. We also varied num_estimators from 100 to 2000. The results are outlined below in figure 6 below.

The best parameters gave us an accuracy of 0.919 on a test split. It was difficult to experiment with larger trees as they take up more space than we had resources for, even with re-sizing the data. As we increase the value of max_features from $\log_2$ to 500, the validation accuracy first increases and then decreases. This can be explained by the fact that the generalization error of random forest depend on both the correlation between any two trees and the strength of each individual tree. As we increase the number of features used in the splitting, the correlation increases and thus increases the error rate. At the same time, however, the strength of the individual trees also increases which leads to a decrease in the error rate.[6].

VI. Discussion

i. Pre-Processing

Performing preprocessing on the data as we have done in our research helped the performance of the models by reducing the problem to a much simpler "function" for the classifiers to learn. However, possible faults in the preprocessing method occur when the algorithm fails to tell two connecting digits apart and includes them both in the processed image. Although this does not happen frequently enough to drastically change the results, it is possible to be improved in the future to further increase the accuracy of the models to what state of the art implementations currently achieve for the original MNIST problem. Another potential implementation that we thought of was training classifiers on the original MNIST data-set, and once having better preprocessing methods, simply feed the training set provided as validation set to the classifiers and perform hyper-parameter tuning from there. A potential classifier that could perform well here would be the k-NN classifier as the digits in the images provided originate from the MNIST data-set, and there is a theoretical 100% chance of finding the identical set of pixels in the MNIST data-set. However, this involves much more work on the preprocessing to make sure images are rotated and scaled to be straight to look similar to the MNIST digits.

One other area of improvement would be our data-augmentation. Although we have not achieved pleasing results with this method, it is widely known to act as a source of improvement of accuracy in image classification problems. We believe the main pitfall in our implementation of data-augmentation is that once we included augmentation, we only fit the classifiers to the augmented data, and not the original data. It is known that for instance if a model sees all digits at the center of the image during training, once shown a small digit at the corner of an image it would not be able

to classify it well. The opposite of this applies as well, if our model has only seen augmented data during training and not the original training data, then once shown centered test data (resulting from our preprocessing methods) it would not be able to perform well on it. Perhaps a combination of augmented and original data would perform best in this case.

ii. Pros/Cons of methodology

In several of our models we opted for some sort of feature reduction. In the case of SVMs and Random Forests, we trained on 28x28 images scaled down. This loss of resolution may have caused small decreases in accuracy as we lose detail, but also made it possible to run a much more thorough GridSearch and try more complicated models. In the case of K-NN, PCA helped reduce runtime significantly without a noticeable loss in accuracy.
If we had a more powerful computer or would be able to scale the images down even further an attempt at SVM with a non linear kernel would likely prove fruitful.

VII. Statement of Contributions

For the implementation, Charlotte Ding has worked on data-augmentation for pre-processing and the complete implementation of the Random Forests classifier. The implementation of CNN was a collaborative work between Nima and Charlotte. The NN was implemented by Nima, as well as the CNN architecture design. Linear SVM and k-NN has been implemented by Pavel, as well as running all the models on his GPU equipped PC. Generating the statistical diagrams shown in the report as well as the Appendix section was also done by Pavel. Hyper-parameter tuning on all models was equally collaborative effort in terms of design but run by Pavel. The report was worked on equally by the three of us.
We hereby state that all the work presented in this report is that of the authors.

References

[1] MNIST. *Modified National Institute of Standards and Technology database* is a large database of handwritten digits that is commonly used for training various image processing systems available here: `http://yann.lecun.com/exdb/mnist`.

[2] *scikit-image* is a collection of algorithms for image processing. It is available to use for free here: `http://scikit-image.org/`

[3] *Artificial Neural Network* `https://en.wikipedia.org/wiki/Artificial_neural_network`

[4] *A visual proof that neural nets can compute any function* By Michael Nielsen / Dec 2017: `http://neuralnetworksanddeeplearning.com/chap4.html`

[5] *Neural Networks and Deep Learning* by Michael Nielsen, 2017. `http://neuralnetworksanddeeplearning.com/index.html`

[6] *Machine Learning* by Breiman, L., 2001. `https://doi.org/10.1023/A:1010933404324`

[7] *The Elements of Statistical Learning (Second Edition)* By Trevor Hastie, Robert Tibshirani, and Jerome Friedman, 2008.

[8] *Keras* The Python Deep Learning library `https://keras.io`

VIII. APPENDIX