# Predicting Unit-Test Scores for Graphic Representations of Simple Computer Programs

Richard Lee Davis
Stanford University
520 Galvez Mall, Stanford CA 94305
rldavis@stanford.edu

## 1. Introduction

Computer science and programming play a fundamental role in our society. There are few aspects of everyday life that remain untouched by software. We communicate over email and VoIP, drive in cars with computer-controlled brakes and engines, relax by playing games on phones, and navigate through the city using GPS systems. This technological explosion has created an urgent need for more programmers. In response to this need, enrollment in computer science courses has ballooned [2]. However, even with increased enrollments, there are fears that the US may be short 1,000,000 programmers by 2020 [7].

Consider two approaches to this issue: one practical, the other aspirational. A practical approach to the issue of needing more programmers has been to broaden computer science education to a wider audience using online courses, coding bootcamps, and increased enrollments at traditional institutions. However, with massive increases in the number of students enrolled in a single course it has become difficult to provide personalized, meaningful feedback and guidance. There are simply not enough qualified teachers to work with every student. One way addressing this has been to build intelligent systems that are capable of providing personalized, meaningful feedback automatically.

An aspirational approach to this issue has been to focus on automating computer programming. If software can be created to write new software, or even to provide engineers with assistance in writing new computer programs, the needs for 1,000,000 new programmers can be met without drastically expanding the computer science education infrastructure. This is not a new goal—in 1958 Lisp was designed in part to make meta-programming easier—but it remains an important one.

Although it may appear that these two approaches have little in common, neither can succeed without one thing: a way of representating computer programs that can be ingested and understood by another piece of software. For an autonomous feedback system to provide meaningful guidance, it must be able to read in and understand student code to the degree that it can provide suggestions for code changes, identify bugs, and help solve problems. And likewise, an automated programming system should be able to learn by ingesting existing, well-written code, and without a way of representing code this is an impossible task.

This paper explores the effectiveness of representing simple programs as images that can be used to train convolutional neural networks. We worked with the code.org dataset, which contains over 100,000 unique code submissions encoded as abstract syntax trees (ASTs) for two distinct coding puzzles (HOC4 and HOC18). Each AST was paired with a unit-test score from -1 to 100 which we used as supervised labels. We developed six related methods for encoding abstract syntax trees as images, and compared the effectiveness of several different convolutional architectures at predicting the unit-test scores using the different types of images as data. We achieved a 15% improvement over a baseline model on HOC4, and a 40% improvement over the baseline on HOC18. Further examination shows that the neural architecture is able to achieve consistently high performance on different graphical encodings as long as the original AST and image are isomorphic, but struggle to beat baseline once this isomorphism is broken.

## 2. Background

### 2.1. Program Representations

The ability to represent programs in a way that can be passed as data into an automated reasoning system is an active area of research. When this is the goal, a natural starting place is the abstract syntax tree (AST). The abstract syntax tree representation generalizes across different languages, and perfectly captures the program's functionality. This representation has been used successfully for detecting redundant code in large code bases [1, 12], for detecting similar code across different languages [3], and for tracing the evolution of code over time [8].

Although operating directly on the AST for a program is useful for detecting similarity between different programs, there are situations where the raw AST may be too complex of a representation to be useful. More recently, researchers have attempted to overcome this limitation by preprocessing ASTs in various ways. Nguyen et al. grouped ASTs of programs into semantic equivalence classes [9], where each semantic equivalence class contained ASTs with different structure but identical function.

An alternative approach to reducing the complexity of the AST is inspired by recent work in natural language processing [13, 14] that recursively parses natural language input and learns vector-space representations of the full text. A variant of this method is employed by Piech et al. to construct dense vector representations of ASTs that group naturally based on functionality [10].

In this paper, we embed ASTs in images, then to use a variety of deep CNN architectures to classify the images based on their unit-test scores. This work is inspired by prior work on image captioning using CNN architectures [15, 11]. CNNs have been able to achieve state-of-the-art performance on captioning images; a complex task that requires understanding the semantic structure in an image. Here, we attempt to use the same convolutional architectures that succeed in image captioning to learn the semantic structure in a visual representation of an AST.

## 3. Methods

### 3.1. Data

The data was derived from code.org dataset (https://code.org/research). This code.org dataset consists of user interaction data for visitors on the code.org website who worked on two different programming challenges. The first programming challenge, HOC4, required users to build a simple program to help a bird navigate a maze (Figure 1). The second programming challenge, HOC18, was more complex, requiring the use of conditional statements and loops (Figure 2). The data contains 3630 unique HOC4 ASTs and 56112 unique HOC18 ASTs from 762,974 unique users. Each code submission is saved as an AST in a single JSON file. In addition, the unit-test scores of each AST are provided.

We trained models on each problem separately. To do so, we separated the HOC4 data into a training set (60%), a validation set (20%), and a test set (20%), and separated the HOC18 data into a training set (60%), a validation set (20%), and a test set (20%).

### 3.2. Creating Graphical Representations of Abstract Syntax Trees

Since the raw data provided by code.org only contained ASTs encoded as JSON objects, it was necessary for to transform the ASTs into images. We wrote a small program

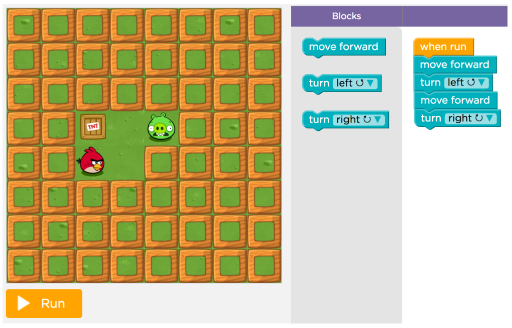


Figure 1: The code.org graphical programming environment, challenge HOC4. Users drag programming blocks onto the canvas to solve the challenge.

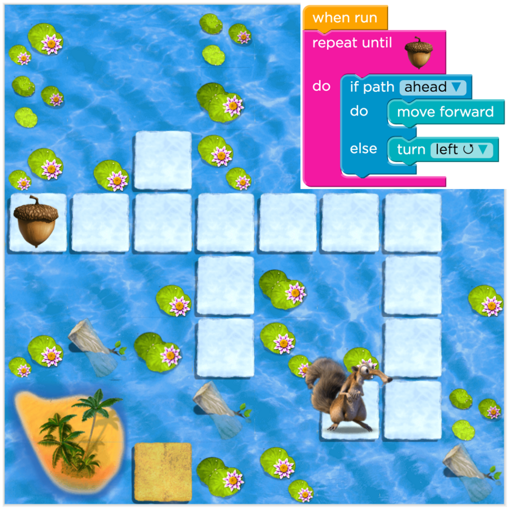
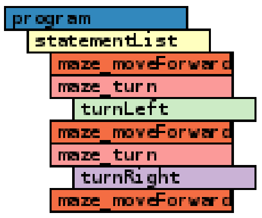


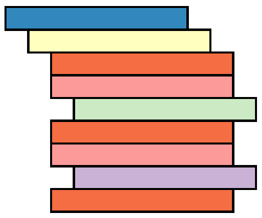
Figure 2: The code.org graphical programming environment, challenge HOC18. This problem is more complex than HOC4, requiring the use of a loop and a conditional.

to ingest JSON files and output images 3. Each AST was transformed into four distinct types of images. All output images were RGB 250px by 250px. After training models on each type of image, it was possible to compare each architecture on each type of image. This allowed us to learn more about which features of the image (e.g., indentation, color) were being used by the model to make predictions.

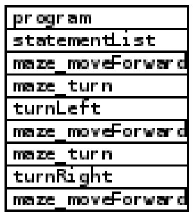
In the two image categories containing indentation, each level of indentation represented a deeper level in the AST. This choice was inspired by common pseudocoding conventions and the Scratch programming language, which uses a similar graphical representation for programs [6]. Each unique expression in the language was assigned a distinct color, with colors chosen using the ColorBrewer2 software for maximal contrast (colorbrewer2.org). An example of how the JSON file was transformed into a full-color, text, indented image can be seen in Figure 4.



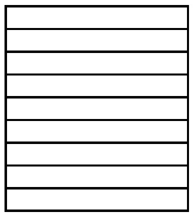
(a)



(b)

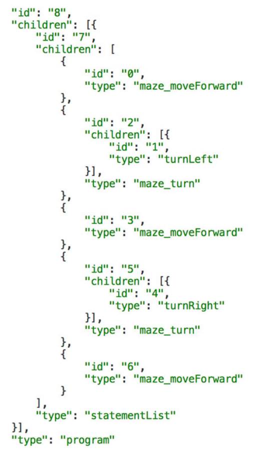


(c)



(d)

Figure 3: The four types of images generated from code.org ASTs. (a) Full-color, text, and indentation. (b) Full-color, indentation, no text. (c) Text-only. (d) Blank



```
"id": "8",
"children": [{
    "id": "7",
    "children": [
        {
            "id": "0",
            "type": "maze_moveForward"
        },
        {
            "id": "2",
            "children": [{
                "id": "1",
                "type": "turnLeft"
            }],
            "type": "maze_turn"
        },
        {
            "id": "3",
            "type": "maze_moveForward"
        },
        {
            "id": "5",
            "children": [{
                "id": "4",
                "type": "turnRight"
            }],
            "type": "maze_turn"
        },
        {
            "id": "6",
            "type": "maze_moveForward"
        }
    ],
    "type": "statementList"
}],
"type": "program"
```

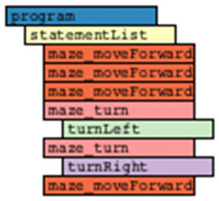
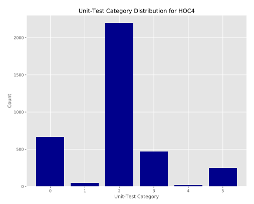


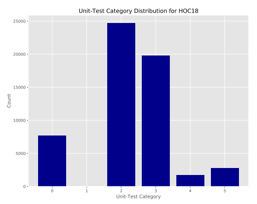
Figure 4: An example AST with the full-color, text, indented image generated from that AST.

## 3.3. Grouping Continuous Unit-Test Scores into Categories

Each AST was paired with a unit-test score ranging from -1 (i.e., the character ran into the wall) to 100 (i.e., the character reached the goal with the minimal number of expressions). We grouped these unit-test scores into six categories, which allowed us to use cross-entropy loss to improve model stability. The categories were as follows:



(a)



(b)

Figure 5: The distribution of classes in HOC4 (a) and HOC18 (b).

- Category 0: The character ran into the wall
- Category 1: The character did not move closer to the goal
- Category 2: The character moved less than halfway to the goal
- Category 3: The character moved more than halfway to the goal
- Category 4: The character made it to the goal with many extra expressions
- Category 5: The character made it tot he goal with a small number of expressions

The distribution of categories can be seen in Figure 5. The non-uniformity of the distribution was partially addressed in the loss function.

## 3.4. Models

We initially trained a number of different model architectures on a subset of the HOC4 data to identify the models that were able to achieve good performance. We settled on a model of our own design (Vanilla CNN), two ResNet architectures (ResNet-18 and ResNet-50) [4], and a SqueezeNet architecture (SqueezeNet 1.1) [5]. These models trained quickly while achieving prediction accuracy on par with larger, slower models like VGG. Finally, we included a baseline model that predicted the most-common class in the data. The only model with undocumented architecture is the Vanilla CNN model. This model contained a single convolutional layer with a 12x12 kernel with stride 2 and a ReLU nonlinearity and batch normalization, followed by three fully-connected layers (size 7500, 300, and 300) each with dropout of 0.3. All models used weighted cross-entropy loss, with decreased emphasis on categories 2 and 3 which were over-represented in the data (Figure 5).

## 4. Experiments

We conducted two full experiments, one on the HOC4 data and one on the HOC18 data. Each experiment had the following goals:

- Achieve the highest possible accuracy on the full-color, text, and indentation images
- For each model, compare accuracy across the four different image classes

In both experiments, we found that three out of four architectures were able to perform significantly better than baseline. The top-performing model in the first experiment (HOC4) achieved an accuracy of 75% (compared to a baseline of 61%), and the top-performing model in the second experiment (HOC18) achieved an accuracy of 80% (compared to a baseline of 40%). Additionally, we found that two out of four models were able to achieve comparably high accuracy on the text-only images ((c) in 3) and the full-color, indented images ((b) in 3). None of the models were able to out-perform baseline by a significant margin on the blank images ((d) in 3).

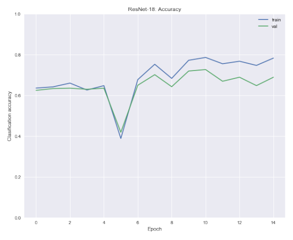
Regardless of architecture, all of the models achieved a similar accuracy. In an attempt to break this ceiling, we compared two models on the HOC18 data: a ResNet-18 model trained from scratch and a ResNet-18 model trained using transfer learning. We found no significant difference in performance between the model trained from scratch and the model trained using transfer learning.

## 4.1. Experiment 1: Predicting Unit-Test Scores for HOC4

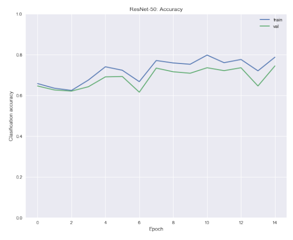
In this experiment we trained four model architectures (Vanilla CNN, ResNet-18, ResNet-50, and SqueezeNet 1.1) on the HOC4 data. We first trained models on the full-color, text, indented images. These images contained the most information and we hypothesized that the various models would achieve their highest accuracy on them. Next, we trained these same architectures on three other types of images. Each type of image removed one or more pieces of information from the full-color, text, indented images. For example, the full-color, indented images removed the text printed inside of each block (compare (a) to (b) in Figure 3).

### 4.1.1 Full-color, text, and indentation

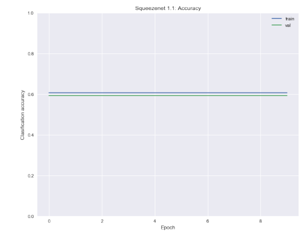
We trained four different model architectures on the full-color, text, indented images for 10 epochs. To compare models, we made predictions on the validation set. The baseline validation accuracy on the HOC4 data was 59%. Both ResNet architectures achieved performance well above baseline: ResNet-18 achieved a validation accuracy of 73% and the ResNet-50 achieved a validation accuracy of 74%. However, neither the Vanilla CNN nor the SqueezeNet 1.1 were able to beat baseline (Figure 6). Note that the SqueezeNet 1.1 failed to improve its performance across 10 epochs.



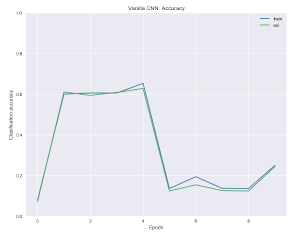
(a)



(b)



(c)



(d)

Figure 6: Comparing the accuracy curves for the ResNet-18 (6a), ResNet-50 (6b), SqueezeNet 1.1 (6c), and Vanilla CNN (6d).

To better compare model performance, we re-loaded the weights for each model that achieved the best validation accuracy, made predictions on the validation set, and looked at each model's confusion matrix. We discovered that both ResNet models made similar predictions. They were both good at correctly classifying images from categories 2 and 3, while performing poorly on category 0. Because categories 2 and 3 contained the majority of the data, by predicting those two categories correctly both ResNet models were able to soundly beat the baseline. The SqueezeNet 1.1 model got stuck in a local optimum: always predicting category 2. The Vanilla CNN was somewhere between the SqueezeNet and the ResNet in terms of its classification performance (Figure 7).

### 4.1.2 Comparing model performance across image categories

After determining that some of the models were able to learn to predict unit-test scores with significantly higher accuracy than baseline, we tested two of the models on three other types of images. Each of the image types was created by removing some information from the full-color, text, indented images.

The first type of image we tested our models on were the no-text images. The text labels inside each box were removed, but all of the color and indentation information remained (Figure 3b). We trained two models on these images: the ResNet-18 and the Vanilla CNN. The ResNet-18 achieved a validation accuracy of 72% and the Vanilla CNN achieved a validation accuracy of 0.66%. These accuracies

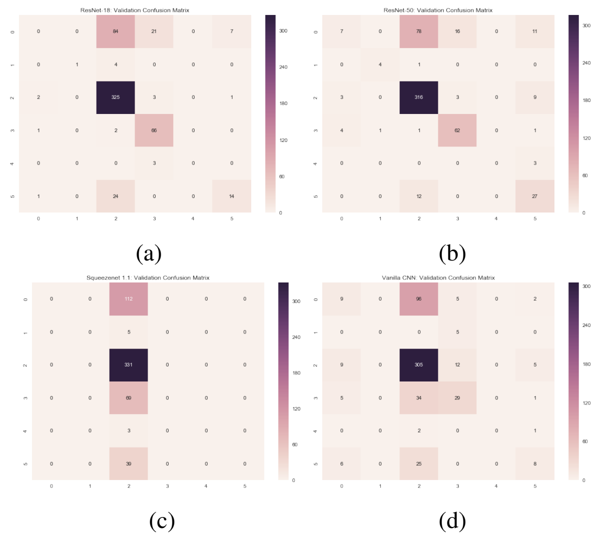


Figure 7: Comparing the accuracy curves for the ResNet-18 (7a), ResNet-50 (7b), SqueezeNet 1.1 (7c), and Vanilla CNN (7d).

were roughly equivalent to the two models' performance on the full-color, text, indented images. The validation accuracy curves and confusion matrices for these two models on this data can be seen in Figure 8.

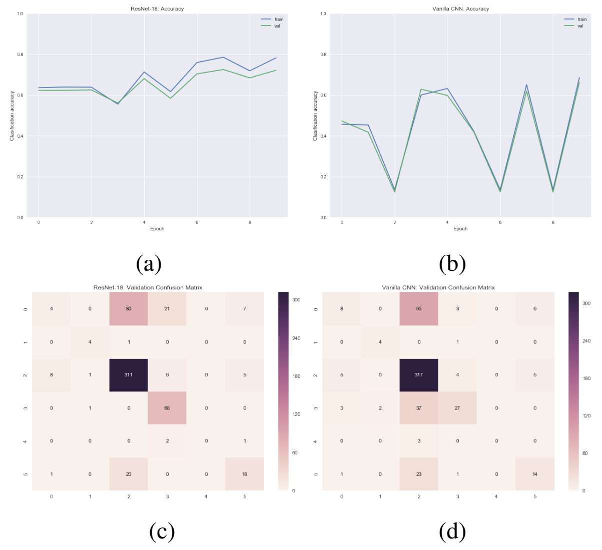


Figure 8: Comparing the accuracy curves and confusion matrices for the ResNet-18 (8a, 8c) and Vanilla CNN (8b, 8d).

The second type of image we tested our models on were the text-only images. These contained no color or indentation information. The images were little more than white blocks with text labels stacked on top of one another (Figure 3c). We trained two models on these images: the ResNet-18 and the Vanilla CNN. The ResNet-18 achieved a validation accuracy of 71% and the Vanilla CNN achieved a validation accuracy of 66%. Again, these accuracies were roughly equivalent to the two models' performance on the full-color, text, indented images. The validation accuracy curves and confusion matrices for these two models on this data can be seen in Figure 9.

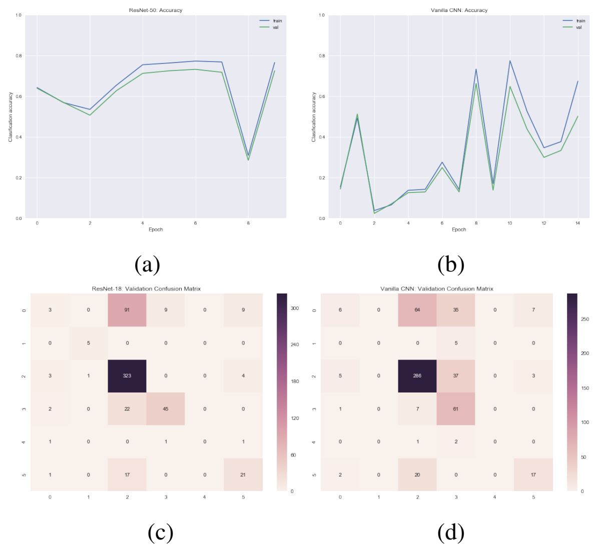


Figure 9: Comparing the accuracy curves and confusion matrices for the ResNet-18 (9a, 9c) and Vanilla CNN (9b, 9d).

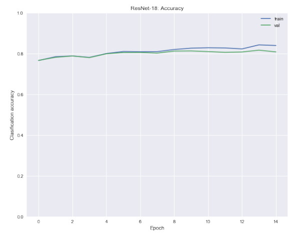
### 4.2. Experiment 2: Predicting Unit-Test Scores for HOC18

In this experiment we trained four model architectures (Vanilla CNN, ResNet-18, ResNet-50, and SqueezeNet 1.1) on the HOC18 data. We first trained models on the full-color, text, indented images. Next, we trained these same architectures on three other types of images. Finally, we compared a ResNet-18 model trained using transfer learning to one trained from scratch.

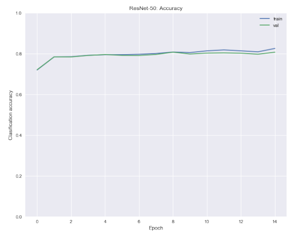
#### 4.2.1 Full-color, text, and indentation

We trained four different model architectures on the full-color, text, indented images for 10 epochs. To compare models, we made predictions on the validation set. The baseline validation accuracy on the HOC18 data was 43%. Again, both ResNet architectures achieved performance well above baseline: ResNet-18 achieved a validation accuracy of 81% and the ResNet-50 achieved a validation accuracy of 80%. However, unlike the HOC4 experiment, the Vanilla CNN was able to perform admirably, achieving a validation accuracy of 79%. Finally, the SqueezeNet 1.1

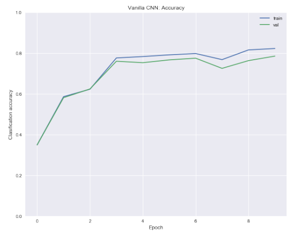
was again unable to beat baseline (Figure 10). As before, the SqueezeNet 1.1 failed to improve its performance across 10 epochs. The validation accuracy plot and confusion matrix are omitted for the SqueezeNet.



(a)



(b)



(c)

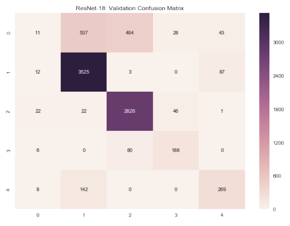
Figure 10: Comparing the accuracy curves for the ResNet-18 (10a), ResNet-50 (10b), and Vanilla CNN (10c).

As in the first experiment, we re-loaded the weights for each model that achieved the best validation accuracy, made predictions on the validation set, and looked at each model's confusion matrix. Although the differences are subtle, the Resnet-18 model appeared to be correctly classifying images from all categories except category 0. The ResNet-50 and Vanilla CNN struggled on categories 4 and 5 more than the ResNet-18. (Figure 11).

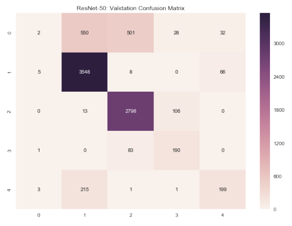
#### 4.2.2 Comparing model performance across image categories

As in the first experiment, we tested two of the models on three other types of images. Each of the image types was created by removing some information from the full-color, text, indented images.

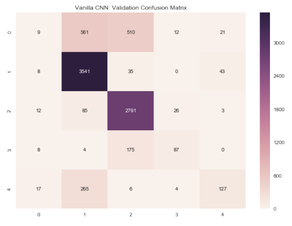
The first type of image we tested our models on were the no-text images. We trained two models on these images: the ResNet-18 and the Vanilla CNN. The ResNet-18 achieved a validation accuracy of 81% and the Vanilla CNN achieved a validation accuracy of 71%. Like the first experiment, the ResNet-18 model was able to achieve identical performance despite the fact that the color and indentation information had been removed. The Vanilla CNN was able to beat the baseline, but was unable to match its performance on the full-color, text, indented images. The validation accuracy curves and confusion matrices for these two models on this data can be seen in Figure 12.



(a)

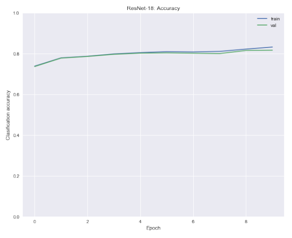


(b)

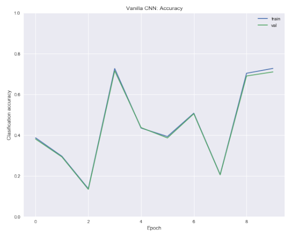


(c)

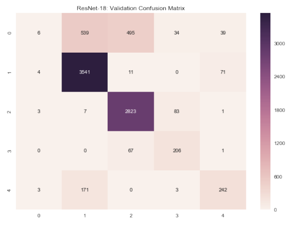
Figure 11: Comparing the confusion matrices for the ResNet-18 (11a), ResNet-50 (11b), and Vanilla CNN (11c).



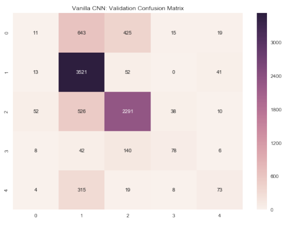
(a)



(b)



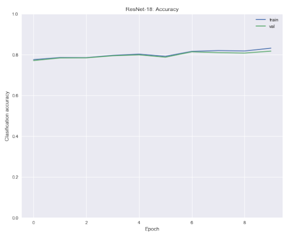
(c)



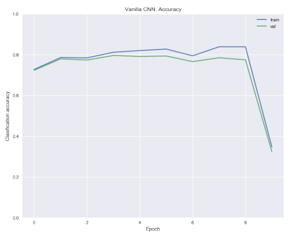
(d)

Figure 12: Comparing the accuracy curves and confusion matrices for the ResNet-18 (12a, 12c) and Vanilla CNN (12b, 12d).

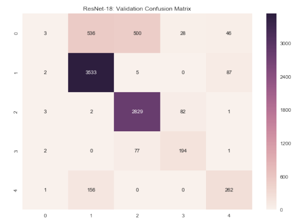
As in the first experiment, the second type of image we tested our models on were the text-only images. The ResNet-18 achieved a validation accuracy of 82% and the Vanilla CNN achieved a validation accuracy of 80%. There was no drop in accuracy for either model despite the fact that all the color and indentation information had been removed. The validation accuracy curves and confusion matrices can be found in Figure 13.



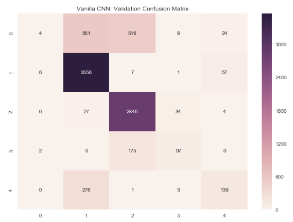
(a)



(b)



(c)

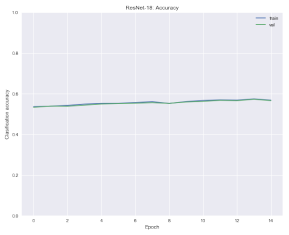


(d)

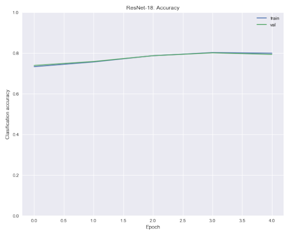
Figure 13: Comparing the accuracy curves and confusion matrices for the ResNet-18 (13a, 13c) and Vanilla CNN (13b, 13d).

### 4.2.3 Comparing Transfer Learning to Training from Scratch

In order to try and achieve an accuracy above 81%, we attempted to use transfer learning. In order to do so, we downloaded pre-trained weights for the ResNet-18 model and only performed gradient descent on the weights in the final fully-connected layer. After training this model for 15 epochs, the accuracy rose from 53% to 57%. We then continued training all of the weights in the model. The accuracy improved from 57% to 81% (Figure 14).



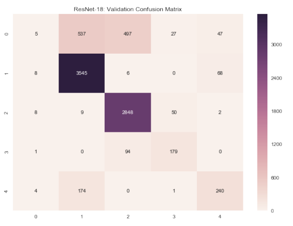
(a)



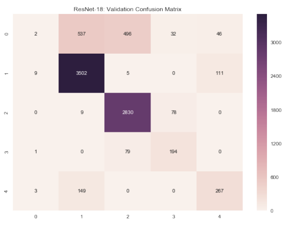
(b)

Figure 14: Accuracy achieved by the ResNet-18 model loaded with pre-trained weights. The accuracy topped out at 57% while only training the final layer (14a) and achieved over 80% when all the weights were trained (14b).

Although the model trained using transfer learning was unable to achieve a higher accuracy than the model trained from scratch, we were curious to know if there was a difference in the distribution of predictions between models. It could have been possible that the model initialized with pre-trained weights would recognize a different set of features, resulting in a different distribution of predictions. However, when we compared the confusion matrices for each model, we found that they were nearly identical (Figure 15).



(a)



(b)

Figure 15: Comparing confusion matrices on the ResNet-18 model trained from scratch (15a) and the ResNet-18 model trained using transfer learning (15b).

## 5. Discussion

Our goal in this study was to determine if it was possible to learn to predict unit-test scores for simple programs if those programs were encoded as images. We ran two experiments on two different coding challenges, and in both cases we found CNNs that were able to significantly beat a baseline. In the first experiment, the best model was able to beat the baseline by 15%, and in the second experiment the best model was able to beat the baseline by nearly 40%. We also performed a set of experiments on images with information removed. We found that even when all indentation and color information was removed from the images, there was little to no degradation in perfromance. Furthermore, when the image and color information was kept and the text information was removed, again there was no degradation in performance. This demonstrated that the CNNs were able to flexibly learn different sets of features (in one case, text, and in the other, color and indentation) with no degradation in performance.

## References

[1] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE. 01188.

[2] T. Camp, W. R. Adrion, B. Bizot, S. Davidson, M. Hall, S. Hambrusch, E. Walker, and S. Zweben. Generation CS: the growth of computer science. 8(2):44–50. 00000.

[3] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 109–118. IEEE. 00625.

[4] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. 02291.

[5] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5mb model size. 00099.

[6] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. 10(4):16. 00515.

[7] C. Mims. Computer programming is a trade; let's act like it. 00003.

[8] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. 30(4):1–5. 00172.

[9] A. Nguyen, C. Piech, J. Huang, and L. Guibas. Codewebs: Scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 491–502. ACM. 00050.

[10] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. Learning program embeddings to propagate feedback on student code. 00022.

[11] M. Ren, R. Kiros, and R. Zemel. Image question answering: A visual semantic embedding model and a new dataset. 1(2):5. 00044.

[12] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer. Detecting similar java classes using tree algorithms. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 65–71. ACM. 00090.

[13] R. Socher, C. C. Lin, C. Manning, and A. Y. Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 129–136. 00584.

[14] R. Socher, C. D. Manning, and A. Y. Ng. Learning continuous phrase representations and syntactic parsing with recursive neural networks. In *Proceedings of the NIPS-2010 Deep Learning and Unsupervised Feature Learning Workshop*, pages 1–9. 00182.

[15] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. pages 3156–3164. 00876.