We thank the reviewers for detailed comments and helpful suggestions. We will incorporate them to improve the paper.

Reviewer 1 — To normalize variable (and function) names, we create a vocabulary of N new identifiers. For each program, we derive a random map from variable names to the new identifiers, and rename all occurrences of a variable with its corresponding identifier. This ensures that the program semantics does not change. We ensure that the size N of our vocabulary is larger than the number of variables in any program in the dataset. This method of reducing vocabulary sizes has been used before in the literature (e.g., [11]).

Network architecture: The reviewer's summary of the embedding layer is correct (L170-L171). We will add more details in the figure as suggested by the reviewer. The first model we tried was more complex and had two more convolutional filters overlapping 2 and 4 rows of the encoded matrix with the strides of 2 and 4, respectively. Through experimentation, we found that removing those filters did not affect the performance of the network but helped in increasing training efficiency. We also tried RNNs (please see response to reviewer#2).

Stride size 3 is used for a convolutional filter spanning 3 rows. So subtree rooted at Node 3 will be analyzed as part of the first 3 rows (Fig 3b). Separately a filter spanning 1 row at a time with stride of 1 (L175-L176) will also cover it.

Comparison with baselines: Given a program, the actual class (success or failure) for a test can be obtained by executing the program. If the classifier predicts success for a test that actually fails, querying the gradients in that case is unlikely to give a meaningful result. Thus, our approach is to be used when the classification is correct and under this setting, we demonstrate that our approach is competitive with (or even better than several configurations of) human-designed SOTA approaches. We will elaborate on this in the paper.

Use of passing tests in Tarantula and Ochiai: Tarantula and Ochiai use coverage metrics over lines (called program spectra), obtained by executing passing and failing tests, and calculate the suspiciousness score for each line based on some empirical formulae. The different configurations in Table 1 indicate how many passing tests were used in the comparison. We will explain this more.

Generalizability to other settings: We exploit similarity between code along with prediction attribution for semantic bug localization and demonstrate it for student code. In the industrial setting also, code similarity abounds due to code reuse and cloning. Large scale studies (see "On the naturalness of software", ICSE'12) have demonstrated that code tends to be quite repetitive (similar to natural language utterances). The practice of version controlling results in similar but evolving copies of code, which are typically subjected to regression testing (L341). Though more experimentation will be required, we expect our approach to be useful in these settings.

Reviewer 2 — Comparison with RNNs: We experimented with an attention-based LSTM network for failure prediction. Training it took more than two days and the performance of prediction attribution was not as good. In comparison, the proposed tree CNN took only one hour to train (L252) and enabled better attribution.

Use of test IDs vs the complete tests: Embedding a unit test along with the program is a great suggestion. This can improve prediction accuracy and attribution, particularly when test code implements some protocols to set up the input objects (such as files). However, in our current setup involving student code, the tests consist of raw inputs and outputs, and lack useful structure. We therefore use only test IDs.

Evaluation section and significance of results: We will reword the evaluation section to make it more clear. Our results show that our technique is competitive to the SOTA dynamic bug-localization techniques which require program instrumentation and collecting program-spectra through multiple executions. We also show that it completely outperforms a naive static approach that uses syntactic difference between a buggy program and its reference implementation for bug-localization.

The percentages shown in the Table 1 correspond to **recall**. **Precision** can be calculated by dividing the number of lines localized by the number of predictions made (= number of programs multiplied by k, where k is the number of suspicious lines reported). Precision values come out to be 0.1, 0.14, and 0.21 when k is set to 10, 5, and 1 respectively.

Scaling to larger programs: We envision the use of our technique at the level of unit tests where methods are tested individually. While method bodies can be large at times, typically they are (encouraged to be) short. Nevertheless, owing to the fast training possible for tree CNN (L252), we are positive about scaling our technique to larger programs.

Reviewer 3 — Classification w.r.t. one test: As the reviewer points out, our classifier analyzes one test at a time. It is an interesting future direction to do localization using entire test suites instead. The dataset-level attribution methods (e.g., based on clustering), called global attribution methods, will be useful in this context.

Runtimes for bug localization: It takes us 4.69 seconds for calculating the embeddings for 8086 correct programs across all the programming tasks (0.5 ms per program). For finding attribution baseline and then performing bug-localization through attribution, it takes about 0.67 seconds per program. We will add these to the paper.