ShapeChecker: Inferring and Reasoning about Shapes in TensorFlow

Miranda Edwards
mirae@cs.washington.edu

Daniel Gordon
xkcd@cs.washington.edu

Beibin Li
beibin@cs.washington.edu

ABSTRACT

TensorFlow [1] is a popular open-source machine learning framework with a declarative data flow graph paradigm. A TensorFlow program involves manipulating data in the form of a tensor (multi-dimensional data structure, i.e. n-dimension array), with a corresponding 'shape' property. The shape dictates whether certain operations (e.g. convolution, dot product, etc.) are legal (similar to a type system) and is inferred or checked at runtime. Finding 'shape' operation bugs before runtime would significantly improve programmer's efficiency but is not currently supported by TensorFlow or any third-party tools. An initial survey of Deep Learning Framework users found that only checking shapes at runtime slows development, and the majority of respondents indicated interest in a tool that would solve this problem. We introduce ShapeChecker, a Vim extension which uses type-checking and type-inference techniques to provide faster feedback to developers about the evaluated shapes of variables.

KEYWORDS

TensorFlow, shape, tensor, Python, machine learning, computer vision, programming system

1 INTRODUCTION

Background on TensorFlow Programs

TensorFlow operates under a declarative programming framework: rather than executing immediately, each TensorFlow API call returns an 'operation' node which is added to the underlying TensorFlow graph. When the programmer wants to execute an operation, they call a function named "session.run" on an operation or list of operations. This call finds the minimum subgraph necessary to execute the listed operations and runs each sequentially. For example:

```
# a, b, c, d are tensors (vectors) with shape [4,].
a = tf.constant([1,1,1,1])
b = tf.constant([1,2,3,4])
c = tf.constant([9,8,7,6])
d = tf.add(a, b) # does not execute immediately
e = tf.sub(c, b) # does not execute immediately
res1 = sess.run(d) # evaluates only a, b, and d
 # returns [2, 3, 4, 5]
res2 = sess.run(e) # evaluates only b, c, and e
 # returns [8, 6, 4, 2]
```

When calling session.run, the programmer provides concrete values for all the necessary inputs to the graph. During graph construction, concrete values are unknown, but oftentimes the input shape is known. Special TensorFlow objects known as "placeholders" allow users to represent the shape of an input during the graph construction phase, but do not require real input values until later in the program. The shape of these placeholder input tensors can be specified when they are declared or discovered at runtime based on the input data. To specify an unknown or dynamic dimension in the graph, programmers use the "None" keyword for the specific dimension. These inputs are fed into additional operations such as convolutions and matrix multiplications.

Sample Program Excerpt:

```
ph = tf.placeholder(tf.float32, [None, 224, 224, 3])
    # takes some number of images that are of type
    float32 and shape 224x224x3.
conv1_weights = tf.get_variable('conv1_weights', [5,
    5, 4, 64]) # creates a weight tensor of 5x5
    convolutions for 4 channel inputs and 64 channel
    outputs
conv1 = tf.nn.conv2d(ph, conv1_weights, [1,1,1,1],
    'SAME') # convolves ph with conv1_weights with
    stride 1 and padding SAME.
```

This code is incorrect because the number of channels in the input (3) should match the third dimension of the convolutional weight tensor (4). TensorFlow Runtime Error:

```
Stack trace
...
  File "/usr/local/lib/python2.7/site-packages/
  tensorflow/python/framework/common_shapes.py",
      line 691, in _call_cpp_shape_fn_impl
    raise ValueError(err.message)
ValueError: Dimensions must be equal, but are 3 and
    4 for 'Conv2D' (op: 'Conv2D') with input shapes:
    [?,224,224,3], [5,5,4,64].
```

Although TensorFlow is a Python-based library, Python is only used as a meta language to define a dataflow graph. All TensorFlow's numerical computations are not performed

in Python [17]. This characteristic makes debugging TensorFlow code particularly difficult in Python, because traditional Python analysis tools are not able to check TensorFlow's backend program. Many machine learning packages, such as PyTorch [14], Theano [3], Caffe [9] etc., also define computational graphs which are later executed using high-performance C++ and CUDA code. Because of the shared coding paradigm, developing an approach to shape-check TensorFlow programs is beneficial to many other tensor-type machine learning development tools.

Motivation

As of January 2018, there are more than 4500 questions on Stackoverflow.com for TensorFlow's shape functionality. To find out if there was a real use case for ShapeChecker, we surveyed 23 ML framework users with a variety of development experience with different deep learning frameworks.

Excerpts From Initial Survey.

- 82.6% of these users says they have struggled with "shape" in machine learning packages.
- On a scale of 1 - 7 (very unhelpful - very helpful), 34.8% users our proposed tool would be very helpful (7)
- 82.6% users think the proposed tool would be helpful (score 5-7).
- 87% of users already have their graph construction in one or a small number of functions.

It can also be very useful to view shapes of intermediate computations during the development process. When deciding how large to make a weight tensor, researchers consider the complexity of the learning problem (how many weights must be learned) as well as the computational complexity of the network (how many floating point operations must be run). This information is currently only available to the programmer at runtime, which creates additional friction when developing a neural network architecture. Another common use-case is reimplementing an existing network architecture described in a paper. These designs are often given only pictorially rather than with code or with the exact operations written down, so it may be easier for the programmer to iteratively write a line of code, check that it matches the paper's figure, and repeat.

Determining shapes statically is not an easy task. It requires tracking each variable's state throughout the execution of a program, and often requires specific estimates of shape values in order to be useful. This is especially difficult in dynamic languages like Python. Both static and dynamic analysis suffer from TensorFlow's underlying complexity; TensorFlow contains over 2700 operations which each act as different transfer functions, and it is time consuming to model all of these operations in shape checking programs.

Contribution

We implemented a user-friendly Vim extension that lets machine learning programmers check the validity of their graph construction and examine the shape of the tensor before running the code, and investigate the effectiveness of such a tool. In this work, we have implemented shape inference for a modern machine learning library, translating knowledge from programming languages to the machine learning domain, and markedly improved productivity for machine learning programmers.

2 RELATED WORK

Only a few research groups have studied multi-dimensional array shaped properties in programming languages and systems [8, 16] in other domains, but ShapeCheck is the first to apply shape inference to machine learning and deep learning tools, as most machine learning packages are still quite young. For instance, TensorFlow was released in 2015 [1] while the published shape-related work was published in the 1990s and early 2000s.

Recently, many studies have tried to improve the computational efficiency and usability of machine learning tools [4, 5, 10]. Few have created shape checking tools for modern machine learning software.

WALA team from IBM has just started to create a static analysis tool for TensorFlow's shape problem (https://github.com/wala/ML), but their work is still in progress. Also, WALA requires users to install its package based on Java, which is not native to Python or TensorFlow. Different from WALA, we are building an user-friendly and light-weight shape checker based on dynamic analysis built entirely in Python.

IBM Watson team created an user interface, DARVIZ, to build deep learning models in Python's Caffe package, and their tool is user-friendly and interactive [15]. However, their tool is only able to create deep learning model that reads image or text data. Moreover, DARVIZ is unable to locate shape bugs in their computation graph.

The shape inference used in this work is inspired by both static and dynamic type inference in programming language [2, 6, 7, 13]. ShapeChecker infers the concrete shape property for each tensor instead of inferring abstract types like in [7, 13].

3 SHAPE CHECKER

Shape Inference

We introduce a tool, ShapeChecker, which dynamically analyzes a TensorFlow program (or even a partial program), producing annotations of useful shape information. In the case of an inconsistency of the shapes or ranks, ShapeChecker produces a useful error message at the point where the inconsistency is discovered. In many cases, intermediate shapes

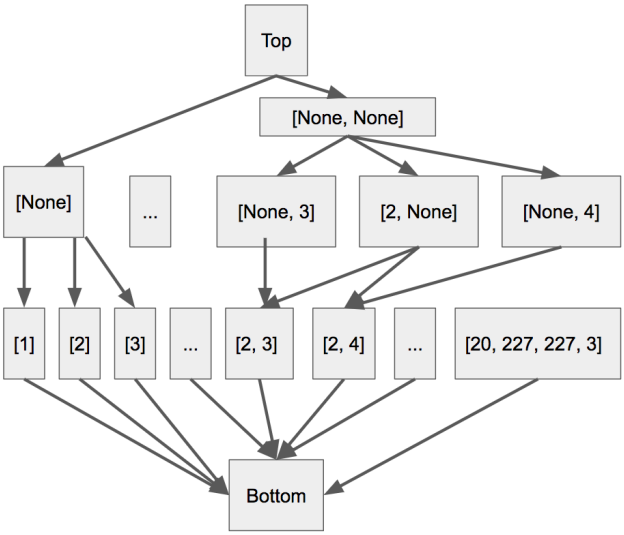
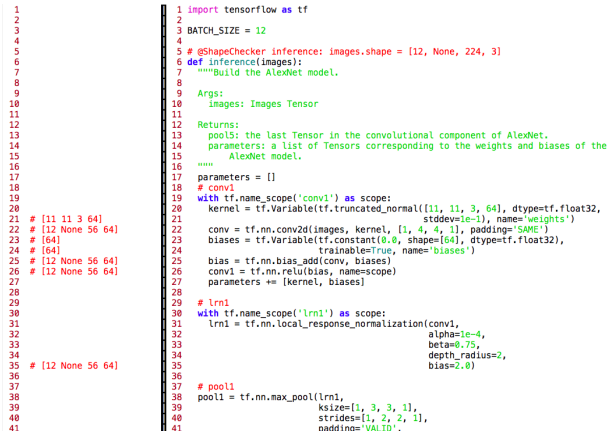


Figure 1: Abstract Interpretation Lattice for TensorFlow Shape. The height of the lattice is bounded by the number of dimensions in the largest possible tensor shape of 256 dimensions.



```
# [11 11 3 64]
# [12 None 56 64]
# [64]
# [64]
# [12 None 56 64]
# [12 None 56 64]

# [12 None 56 64]
```

```
import tensorflow as tf

BATCH_SIZE = 12

# @ShapeChecker inference: images.shape = [12, None, 224, 3]
def inference(images):
  """Build the AlexNet model.

  Args:
    images: Images Tensor

  Returns:
    pool5: the last Tensor in the convolutional component of AlexNet.
    parameters: a list of Tensors corresponding to the weights and biases of the
        AlexNet model.
  """
  parameters = []
  # conv1
  with tf.name_scope('conv1') as scope:
    kernel = tf.Variable(tf.truncated_normal([11, 11, 3, 64], dtype=tf.float32,
                                             stddev=1e-1), name='weights')
    conv = tf.nn.conv2d(images, kernel, [1, 4, 4, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[64], dtype=tf.float32),
                         trainable=True, name='biases')
    bias = tf.nn.bias_add(conv, biases)
    conv1 = tf.nn.relu(bias, name=scope)
    parameters += [kernel, biases]

  # lrn1
  with tf.name_scope('lrn1') as scope:
    lrn1 = tf.nn.local_response_normalization(conv1,
                                              alpha=1e-4,
                                              beta=0.75,
                                              depth_radius=2,
                                              bias=2.0)

  # pool1
  pool1 = tf.nn.max_pool(lrn1,
                         ksize=[1, 3, 3, 1],
                         strides=[1, 2, 2, 1],
                         padding='VALID',
```

Figure 2: Example of the annotation output of ShapeChecker on a TensorFlow program after pressing F-4

are useful feedback for the programmer indicating that the code operates as expected.

ShapeChecker aids in both the debugging and the development process. In the current TensorFlow implementation, checking the shape is done at runtime. Developers have no way of checking their code's correctness other than to run the program. However with lengthy preprocessing or other Python computation, it may take minutes to reach the TensorFlow portion of the code. This slows the programmer down because they must run their program, wait for the preprocessing and TensorFlow code to run, fix a line, then run again from the start. Our tool sidesteps this problem by explicitly running ShapeChecker on the TensorFlow graph construction methods separately from the rest of the program.

User Interface

Our tool contains a Vim-integrated extension to show shape annotation of user's TensorFlow code from an output file generated by ShapeChecker. Our tool also shows the annotations on a line-by-line basis, allowing the developer to glance at ShapeChecker's calculated result without interrupting their thought process. The extension allows the user to press F-4 to open a second window, with comments on each line indicating the resulting shape and whether the line succeeded or failed.

4 METHODS

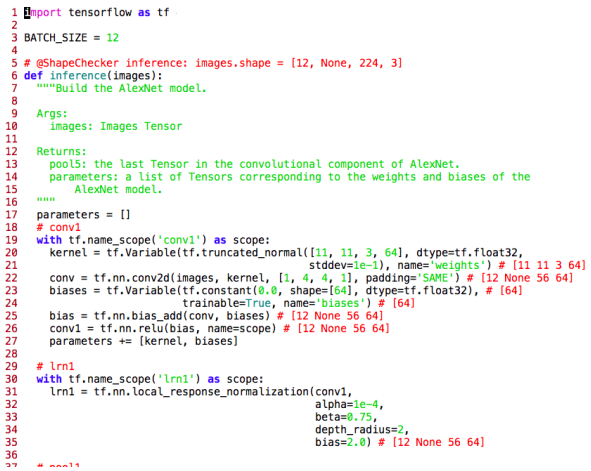
ShapeChecker analyzes Python programs that use TensorFlow by running a modified subset of the code. Users first specify the entrypoint function names and arguments via a specific ShapeChecker annotation comment. Our implementation locates these methods and constructs a mock program that replaces each TensorFlow method call with a mocked method, and runs the modified program. These mocked functions do not run mathematical operations on the values of tensors, they only perform shape-related operations and update the estimate of the current shape value, checking for inconsistencies during the updates.

Modifying the AST

To replace TensorFlow calls with the corresponding ShapeChecker ones, ShapeChecker constructs an AST representing the user's program using Python's standard AST module. This module represents imports as a node-type of the AST, so ShapeChecker is able to simply construct an ImportNode for the ShapeChecker and replace the TensorFlow ImportNode with the ShapeChecker ImportNode. From there, ShapeChecker adds another temporary node that acts as an entry point into the program, handling the construction of any Tensor arguments to match the user-annotated suggested input.

Mocked TensorFlow Functions

The mocked functions operate on (and return) lightweight MockTensors which only hold the shape of the tensor. All TensorFlow graph-related functions return tensor objects (or None), so all of our functions will return MockTensors. Due to time constraints, we chose several hundred (of the several thousand) common TensorFlow functions to mock out. These functions are implemented to match the syntax and behavior of the original TensorFlow function specifications. If ShapeChecker encounters a function which has not been mocked, ShapeChecker exits the analysis phase and returns



```
1  import tensorflow as tf
2
3  BATCH_SIZE = 12
4
5  # @ShapeChecker inference: images.shape = [12, None, 224, 3]
6  def inference(images):
7    """Build the AlexNet model.
8
9    Args:
10     images: Images Tensor
11
12   Returns:
13     pool5: the last Tensor in the convolutional component of AlexNet.
14     parameters: a list of Tensors corresponding to the weights and biases of the
15         AlexNet model.
16   """
17   parameters = []
18   # conv1
19   with tf.name_scope('conv1') as scope:
20     kernel = tf.Variable(tf.truncated_normal([11, 11, 3, 64], dtype=tf.float32,
21                                              stddev=1e-1), name='weights') # [11 11 3 64]
22     conv = tf.nn.conv2d(images, kernel, [1, 4, 4, 1], padding='SAME') # [12 None 56 64]
23     biases = tf.Variable(tf.constant(0.0, shape=[64], dtype=tf.float32), # [64]
24                          trainable=True, name='biases') # [64]
25     bias = tf.nn.bias_add(conv, biases) # [12 None 56 64]
26     conv1 = tf.nn.relu(bias, name=scope) # [12 None 56 64]
27     parameters += [kernel, biases]
28
29   # lrn1
30   with tf.name_scope('lrn1') as scope:
31     lrn1 = tf.nn.local_response_normalization(conv1,
32                                               alpha=1e-4,
33                                               beta=0.75,
34                                               depth_radius=2,
35                                               bias=2.0) # [12 None 56 64]
36
37   # pool1
```

Figure 3: Example of the side-by-side output of ShapeChecker on a TensorFlow program after pressing F-3. The annotation example is shown in line 5.

a warning to the user indicating that ShapeChecker could not process the specific line(s).

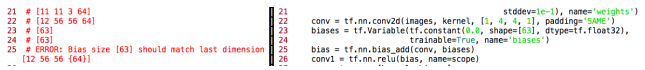
When it is run, ShapeChecker reports errors when a user's code incorrectly calls TensorFlow functions, either with incorrect or invalid shapes of variables (such as multiplying a length 2 vector with a length 3 vector), incorrectly formatted arguments (such as providing a tuple where a string is required), as well as syntax errors in the code. If ShapeChecker finds a shape error, the TensorFlow program is guaranteed to have the same issue.

There are still many TensorFlow errors that we cannot catch. For instance it is illegal in TensorFlow to call "tf.get_variable" twice with the same "name" argument (unless the reuse flag is set). Also, certain operations are only defined on specific numerical types (some arguments are required to be of type Tensor(int), Tensor(float) etc.). We limited the scope of this project to shape errors since our survey noted that shapes were an important issue, and these other error types are usually quite easy to locate and fix.

ShapeChecker Annotations

ShapeChecker requires programmers to indicate which functions they wish to check using specific ShapeChecker annotations, and providing sample input shapes to those functions. This is akin to running a unit test on a subsection of code in a larger project. This paradigm allows the programmer to test their TensorFlow graph construction in an otherwise incomplete program. Based on our initial survey, we have found that the additional overhead to the programmer will not be significant, which is especially important during development.

The ShapeChecker annotations consists of three parts (an example annotation can be seen in figure 2 line 5). The first



```
21  # [11 11 3 64]                                                    21                                              stddev=1e-1), name='weights')
22  # [12 56 56 64]                                                   22    conv = tf.nn.conv2d(images, kernel, [1, 4, 4, 1], padding='SAME')
23  # [63]                                                            23    biases = tf.Variable(tf.constant(0.0, shape=[63], dtype=tf.float32),
24  # [63]                                                            24                         trainable=True, name='biases')
25  # ERROR: Bias size [63] should match last dimension of input [12  25    bias = tf.nn.bias_add(conv, biases)
    [12 56 56 {64}]                                                   26    conv1 = tf.nn.relu(bias, name=scope)
```

Figure 4: Example of an error reported by ShapeChecker

is the signature indicating that a method is meant to be ananlyzed by ShapeChecker. This is always the string literal "@ShapeChecker". The second is the method name for a function that should be checked. The third is an optional set of shapes of any tensors that would be inputs to the function. These annotations are found and parsed using a regular expression search over the current source code file. We require that the checked functions take no additional arguments without default values in order to allow us to test each method independently, similar to unit tests.

Visualizing the ShapeChecker Output

We integrated ShapeChecker with Vim to allow semi-automatic feedback to the developer, with an optional output file. The output file consists of a copy of the source code, with the annotations of the shapes printed in-line as comments. This was designed to aid during the development process without breaking the programmer's workflow.

If ShapeChecker finds a bug in the input TensorFlow program, it reports the bug and source of conflicts causing the bug in the output file, before terminating analysis, as shown in Figure 4 of the appendix.

User Workflow

The expected user workflow is as follows:

(1) A user, in the middle of development, wishes to check that a block of code has been 'shaped' correctly. They can't yet just run the code since they haven't finished writing their program. They break that block off into its own function. At the line where the function is called or above the method declaration (or anywhere within the file), they add an annotation '@Shapechecker function_name' to indicate the entrypoint for ShapeChecker.
(2) They press F-3 or F-4 to run ShapeChecker in Vim, which opens a second window with the output. When F-3 key is pressed, the shape of that variable will be shown together with their original code, as shown in Figure 3. When F-4 is pressed, the output shape will be displayed in the separate window, as shown in Figure 2. The user will see an error message if their TensorFlow program contains shape bugs, as shown in figure 4.
(3) The user can then fix the any shape mistakes if necessary before continuing development.

Layer Type	Units	Filter Size	Stride	Padding
convolution	6	5x5	1x1	SAME
max-pool		2x2	2x2	VALID
convolution	16	5x5	1x1	VALID
max-pool		2x2	2x2	VALID
convolution	120	5x5	1x1	VALID
Fully-Connected	84			
Dropout (0.5)				
Fully-Connected	10			

Table 1: LeNet Architecture

5 EVALUATION

Test Suite

We searched online for available TensorFlow network code in sources such as tutorials, research blogs, official TensorFlow examples etc. and selected 50 programs that are representative of popular Tensorflow uses. Many were from the Zero-to-all TensorFlow tutorial repository https://github.com/hunkim/DeepLearningZeroToAll. We ran ShapeChecker on them, and manually compared ShapeChecker's output with TensorFlow's constructed graph to ensure the accuracy and robustness of ShapeChecker. When using ShapeChecker in the future, we can be confident that it will catch the majority of shape-related bugs.

Questions			
	Q1	Q5	Q6
Score (1-7)	5.4	2.7	
Yes/No (All Yes is 100%)			82%

Table 2: This table summarizes the results from the final survey.

Final Survey

We conducted a user-test where we provided the users with a skeleton version of a program which operates on CIFAR-10 (a popular vision task)[11]. We asked the users to implement a relatively simple network known as LeNet [12], shown in table 1. After they completed the coding problem, we surveyed them. The survey questions are below:

(1) How helpful/unhelpful did you find ShapeChecker: 1-7 (7 is very helpful).
(2) What did you like about ShapeChecker?
(3) What did you not like about ShapeChecker?
(4) What could be added to ShapeChecker that would make it better for you?
(5) Did you find the annotation process burdensome? 1-7 (7 is very burdensome).
(6) Would you use ShapeChecker in the future?

As shown in table 2, the overall feedback was positive and we found that users were much faster at finding, localizing, and fixing bugs. The complete survey responses are located in the appendix.

Based on the survey responses, we added more TensorFlow functions, fixed several bugs, cleaned up the error message handling to better localize issues, and made the ShapeChecker annotation documentation clearer.

6 CONCLUSION AND FUTURE WORK

We have presented a novel tool using type-checking techniques in TensorFlow, a new domain. The advantages of our methodology have been demonstrated by experimental results. Future work would involve expanding the number of TensorFlow functions supported to the complete set of Tensor functions. Other projects might consider porting ShapeChecker to similar ML libraries using tensors such as PyTorch or Keras. Another extension to our project would be to integrate ShapeChecker to other IDEs (e.g. PyCharm, Spyder, etc) and make the use of ShapeChecker more "automatic". Based on the interest demonstrated in the final survey (table 2) in the tool, we plan to add more TensorFlow functions and open-sourced the code on Gitlab.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, and others. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

[2] Alexander Aiken and Edward L Wimmers. 1993. Type inclusion constraints and type inference. In *Proceedings of the conference on Functional programming languages and computer architecture*. ACM, 31–41.

[3] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, and others. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688* 472 (2016), 473.

[4] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, and others. 2013. API design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238* (2013).

[5] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. 2002. *Torch: a modular machine learning software library.* Technical Report. Idiap.

[6] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 207–212.

[7] Philip J Guo, Jeff H Perkins, Stephen McCamant, and Michael D Ernst. 2006. Dynamic inference of abstract types. In *Proceedings of the 2006 international symposium on Software testing and analysis*. ACM, 255–265.

[8] C Barry Jay and Milan Sekanina. 1996. *Shape checking of array programs.* Technical Report. Citeseer.

[9] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[10] Davis E King. 2009. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research* 10, Jul (2009), 1755–1758.

[11] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).
[12] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[13] Robert O'Callahan and Daniel Jackson. 1997. Lackwit: A program understanding tool based on type inference. In *In Proceedings of the 19th International Conference on Software Engineering*. Citeseer.
[14] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).
[15] A. Sankaran, R. Aralikatte, S. Mani, S. Khare, N. Panwar, and N. Gantayat. 2017. DARVIZ: Deep Abstract Representation, Visualization, and Verification of Deep Learning Models. *ArXiv e-prints* (Aug. 2017).
[16] Olha Shkaravska, Ron van Kesteren, and Marko van Eekelen. 2007. Polynomial size analysis of first-order functions. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 351–365.
[17] Bart van Merriënboer, Alexander B Wiltschko, and Dan Moldovan. 2017. Tangent: Automatic Differentiation Using Source Code Transformation in Python. *arXiv preprint arXiv:1711.02712* (2017).

A APPENDIX: EXAMPLE OUTPUT

B APPENDIX: FINAL SURVEY RESULTS

Student 1

(1) 6
(2) I liked the instant feedback that the partial code was right without having to run the rest of the program.
(3) It was kind of brittle. If there was a function that didn't exist, I had to find another.
(4) Add more TensorFlow functions.
(5) 3. I messed up the first time I wrote it and there was no way of knowing what was wrong.
(6) Yes, if it worked on all functions.

Student 2

(1) 6
(2) The output is nice
(3) It's not compatible with Spyder, iPython, or other modern python IDEs.
(4) More information and suggestions for convolution operations. For instance, if it can give suggestions on how large the stride size should be for a specific layer, it would be great. Real time feedback is also important, because sometimes I might forget to press F-4 key.
(5) 3. I don't like it, but it's okay.
(6) Yes, I will use it when writing TensorFlow models.

Student 3

(1) 5
(2) I got fast feedback about the intermediate tensor shapes.
(3) I would have preferred an Emacs plugin. I'd also like error messages when ShapeChecker itself isn't working.
(4) Adding an Emacs plugin.
(5) A little (4). I got the syntax wrong initially and didn't get an error message so I didn't know why it wasn't working.
(6) Yes, especially when extending other people's code.

Student 4

(1) 6
(2) It was really quick and it caught some bugs.
(3) I didn't like when it crashed.
(4) I liked the annotated code better and I didn't want a second window. I want a version that annotates my code (but well formatted). I also want a version that accepts variables in the annotation.
(5) 2.5, I messed it up at the start but it was easy to fix.
(6) Yes, if it's easy to install.

Student 5

(1) 7
(2) It told me the shapes in real time. It gave me fast feedback so I didn't have to think about the shapes myself.
(3) I don't like that it's not in TensorFlow directly. It was also a bit buggy.
(4) Non-vim plugins would be good. I use Jupyter, so I need something that works with that.
(5) 1
(6) Yes, if it worked in my code setup.

Student 6

(1) 6
(2) It lines things up in a nice way that is easy to see. It was simple and easy to use. I especially like the view with annotations on the code (F3).
(3) It was a bit hard to line up the row I actually cared about when I was fixing an issue.
(4) An option to put on the right side so it's at the end of the line of code.
(5) 1
(6) Yes definitely.

Student 7

(1) 3
(2) It was nice to have the size right in the code while I was writing it.
(3) It doesn't work on some functions.
(4) Integrate with PyCharm. Add on-the-fly commenting (it runs as I type a line) or put the comments in the same file.
(5) 1
(6) Yes, why not. It was helpful.

Student 8

(1) 6

(2) It was a shortcut to how I normally do debugging.
(3) It kills undos. Errors weren't always obvious why they were happening, if it was my code or somewhere else.
(4) Having provenience would be cool.
(5) 2
(6) No because I don't like maintaining plugins

Student 9

(1) 5
(2) It is helpful to debug the dimensions in tensorflow
(3) N/A. There is no inconvenience, and I can always choose to ignore the ShapeChecker result.
(4) If it can show the (hidden) parameters for each function call, it will be more helpful for debugging.
(5) 3
(6) Yes.

Student 10

(1) 4
(2) I could see how it's useful to know about bugs before finishing
(3) The plugin output was kind of hard to read, and was pretty buggy.
(4) If it was more automatic somehow.
(5) 5
(6) Probably not.

Student 11

(1) 5
(2)
(3) It took some time to figure out how to use it, and I had to factor out my code more than I would've otherwise.
(4)
(5) 4
(6) Sure. It'll probably be easier to use as I use it more.