Onion Language Reference Manual

By: Andrew Aday, (aza2112) Amol Kapoor (ajk2227), Jonathan Zhang (jz2814)

Introduction

In this document we propose *Onion*, a new language developed to make the design and implementation of deep learning models easier and more intuitive. Deep learning is an extremely popular field that is quickly becoming a mainstay of major tech companies like Google and Amazon. Deep learning models consistently beat state of the art algorithms in many fields. Deep learning models are mostly created with libraries built on top of Python, e.g. Tensorflow. Though powerful, these libraries lack visual clarity and are in general cluttered by other unnecessary features coming from a general purpose language like Python. *Onion* is a language built from the ground up specifically for deep learning models that emphasizes visual clarity and minimalism. We believe *Onion* can help deep learning experts program deep learning models the way they think about deep learning models, leading to fewer errors and less programming time.

Onion aims to be a layers-based library like Slim and Keras, optimized for complex deep learning models. *Onion* optimizes for deep learning through *Onion's* unique syntax: almost everything in the language is defined as a layer (processing) or as an array (data). The minimalist nature of the language makes it extremely easy to quickly define layers that stack into full, complex deep learning models. The language also allows users to quickly identify how a data flows through a model (i.e. what components of a piece of code are processing and what components are data).

Types

Type	**Description**	**Example**
`int`	Integer Data Type	`int i = 5`
`float`	Float Data Type	`float f = 5.0`
`bool`	Single byte boolean (0 or 1)	`true`
`string`	String Data Type	`“Hello world”`
`fmatrix`	nxn float matrix type - default	`fm = fmatrix([10, 10], 1)`
`imatrix`	nxn integer matrix type	`imatrix im = imatrix([10, 10], 1)`
`smatrix`	nxn string matrix type	`smatrix sm = [[‘hello’], [‘world]]`

Lexical Conventions

Identifiers

Variables must begin with a lowercase character, and may contain only a combination of lowercase characters and underscores (“_”). “_” is not a valid variable name. Variables should be snake case.
Models and Layers must begin with a capital character, and may only contain a combination of capital and lowercase layers. Models and Layers should be camelcase.

Keywords

Keyword	**Description**
`for`	For loop
`if`	If-elif-else block
`elif`	If-elif-else block
`else`	If-elif-else block
`return`	Return function expression
`no`	Void return type
`model`	Declare a model object
`layer`	Declare a layer object
`optimizer`	Declare an optimizer layer object
`func`	Declare a function
`batch`	Declare a batch object
`trainable`	Declare a trainable object
`train`	Keyword used to indicate when a model is training

Comments

/* I am a comment */

```
/*
        And I am a blocky comment
*/
```

Expressions

Primary Expressions

Identifier: Variable, function, and model names
Constants:

Constant	**Example**
int	5
bool	true or false
float	5.0
string	“Hello world!\n”. Note how special characters are prefixed with a backslash, as in c.

Unary Operators

(-expression): defined for int and float. Evaluates to the negative value.
(!expression): logical negation. Booleans only.

Arithmetic Operators

If either expr1 or expr2 is type float, arithmetic operators will cast the other expr to float and return a float.

(expr1 + expr2): sum float or int, concatenate strings.
(expr1 - expr2): substract floats or int.
(expr1 * expr2): multiply floats or int
(expr1 / expr2): divide floats or int
(expr1 % expr2): modulo ints
(expr1 ** int expr2): exponentiation. Note expr2 must be an integer type

Matrix Operators

(matrix1 ^* matrix2): matrix multiplication on fmatrices or imatrices
(matrix1 ^.* matrix2): hadamard product fmatrices or imatrices. Matrices must be same shape
(matrix1^): transpose

Assignment Operators

(expr1 = expr2): sets the value of expr1 equal to the value of expr2. Both expressions must be of the same type; Onion does not do implicit type conversion during assignment.
(expr1 => expr2): sets the value of expr2 equal to the value of expr1
See **Syntax and Program Structure: Assignment** for more details.

Relational Operators (returns bool)

(expr1 < expr2) less than
(expr1 > expr2) greater than
(expr1 <= expr2) less than or equal to
(expr1 >= expr2) greater than or equal to

Equality Operators (returns bool)

(expr1 == expr2) equality
(expr1 != expr2) inequality

Logical Operators (returns bool)

(expr1 and expr2) logical and
(expr1 or expr2) logical or

Syntax and Program Structure

Assignment

Normal declaration and assignment

Declaration of variables is done by indicating the type of the variable followed by a variable name (in snake case). If the variable being defined is a fmatrix, it does not need to have a variable type, although it may. More explicitly: <u>variables without a declared type are assumed to be of type fmatrix.</u> We do this because fmatrix is expected to be the most common type.

Assignment can be done with the equals (=) operator. The Onion language uses left operand assignment, i.e. the identifier on the left of the equals operator is set to the expression on the right of the equals operator. For example:

```
int x = 5;
int y = 10;
Print(x + y, “%d”); /* Prints out 15 */

x = [0.1, 0.2, 0.3]
Print(x[0], “%f”); /* Prints out 0.1 */
```

Pipe assignment

The model section of an Onion program can assign variables using the pipe (=>) operator. The pipe operator takes the value of the left operand and ‘pipes’ the value into the right operand (which may be on a newline). If the pipe operator is used with a Layer or Model component as the left operand, it will take the return of the Layer or Model (and throw an error if no return is defined). If the pipe operator is used with a Layer or Model as the right operand, it will pipe the value of the left operand as the first parameter for the Layer or Model on the right. For example:

```
int x = 5;
x =>
int y =>
Print("%d”); /* Prints 5 */
```

Matrix declaration

Matrices can be declared with initial values using brackets. Each value in the matrix must be of the declared type. For example:

```
imatrix x = [[0, 1, 2], [3, 4, 5]]; /* defines 2x3 int matrix */
```

Tuple declaration and assignment

Tuples are a unique data type that are used exclusively for piping data into Layers and Models. A tuple syntax uses optional parens around comma separated variables of various types and shapes. For example:

```
layer Sum((int x, int y, float z)) int {
      Print(z, “%f”);
      return x + y;
}

int x = 5;
int y = 10;
float z = 0.5;
x, y, z => Sum();  /* parens are optional */
(x, y, z) => Sum();
```

Control Flow

Statements and blocks

Statements will be delineated by a semicolon:

```
int statement = Statement();
```

Blocks will be delineated by opening and closing curly brackets. Variables are local to the nearest enclosing block.

```
int a = 1;
{
      int a = 5;
      a => printf("%d\n");  /* prints 5 */
}
a => printf("%d\n");  /* prints 1 */
```

The keyword `for` is used for both for-loops and while loops.

```
for (int i = 0; i < 5; i = i + 1) {  /* for loop */
      ...
}

int i = 5
for (i < 5) {  /* while loop */
      ...
      i = i + 1;
}
```

If-elif-else blocks are standard. You must delineate the statements with curly brackets {}. I.e. one-line if statements without curly brackets are NOT allowed.

```
if (condition) {
} elif (condition) {
} else {}
```

Anonymous functions

Anonymous functions are only allowed within the scope of our predefined `Map()` function. They are limited to a single statement, and the result of that statement is used as the return value. Map can call a `func` if multiple statements are needed. Anonymous functions have the following syntax:

```
(float f): f + 1.0  /* single statement anonymous fn */
(float f): foobar(f)  /* multi-statement anonymous fn */
```

Predefined Components

Component Name	Example
print(str)	`print("hello world!");`

printf((arg1,arg2,...) , formated_str)	`printf((1,2), "%d and %d");`
	`/*1 and 2*/`

map(matrix, function)	`map(`
	`    [1,2,3],`
	`    (int i): i + 1`
	`) /* returns [2,3,4] */`

A Note on Keywords

The use of some keywords may not be immediately clear. This section serves to clarify how these keywords work.

batch

The `batch` keyword is used in the parameters list of models. The keyword indicates when an input parameter should be treated as batched, i.e. the model treats the first dimension as a collection for the data stored in the rest of the dimensions. `batch` can only be used as a tag for matrix types (fmatrix, imatrix, smatrix). Any parameter set with batch must have the same first dimension size. If at least one parameter is set with the `batch` keyword, the `batch_size` parameter can be set. `batch_size` indicates how many elements of the first dimension of a batch should be used per batch. `batch_size` must be positive, less than the size of the batch input, and divide the batch input cleanly. A few examples:

```
imatrix x = [[0, 1, 2], [3, 4, 5]];
```

```
model BatchExample(batch imatrix x) {
      /* x is treated here as a 1x3 matrix, i.e.
         [0, 1, 2] and [3, 4, 5] are passed in separately and loss is
         collected from the batch (the batch size defaults to the size of
         the input, i.e. 2)*/
}

model BatchSizeExample(batch imatrix x, int batch_size=1) {
      /* x is treated here as a 1x3 matrix, i.e.
         [0, 1, 2] and [3, 4, 5] are passed in separately and loss is
         collected from the batch, in this case from each input */
}

model NonBatchExample(imatrix x) {
      /* x is treated here as a 2x3 matrix, all of the data is passed
         together */
}
```

trainable

The **`trainable`** keyword is used to indicate which variables in a layer can be modified by optimizers (i.e. can be changed by model training). Any variable that is tagged with **`trainable`** can be accessed by optimizer layers. The **`trainable`** keyword can only be used for fmatrices and only used within layers.

train

The **`train`** keyword is used to indicate when a model is training. A model that is activated in **`train`** brackets runs any optimizer layers. A model that is activated outside of **`train`** brackets does *not* run the optimizer layers. **`train`** brackets can only be used in the run section of the Onion program.

Program Sections

Functions

Functions are extremely basic numerical computation components in the Onion language. Unlike the other sections listed below, functions are entirely optional and do not need to be used. An example function may look something like this:

```
func Max(int x, int y) int {
      if (x > y){
            return x;
```

```
    }
    return y;
}
```

The `func` keyword *must* be followed by a capitalized string name composed of only upper and lowercase alphanumeric characters. After the parameter list, the function states its return type. If no return type is specified, it is assumed to be `fmatrix`. The function definition *cannot* use any of the unique Onion keywords or operators defined below -- it *cannot* use the pipe operator, the trainable keyword, the batch keyword; it *cannot* call other layers, models, or functions (and therefore *cannot* recurse); it *cannot* use the train tag or FileIO.

Layers

Modular abstraction is one of the core ideas of the Onion language. Layers are meant to be used for manipulating float values in fmatrices directly. They therefore allow users to define individual components of a network that can be inserted and reused throughout a model definition. As with functions, layers must state their return type after the parameter list. If no return type is provided, `fmatrix` is assumed. Relu layers and FC layers are both defined below.

```
layer Relu(inputs) {
    Map(inputs, lambda x: Max(x, 0));
    return inputs;
}

layer FC(inputs, int output_dim, [, str name]) {
    trainable weights = fmatrix([inputs.shape, output_dim], 1);
    return inputs ^* weights;
}
```

The `layer` keyword *must* be followed with a capitalized string name composed only of upper and lowercase alphanumeric characters. A layer definition *cannot* call another layer or model, and *cannot* use the pipe operator. A layer definition *can* index inputs and manipulate individual float values. A layer definition *can* use the `trainable` keyword to signify the creation of graph variables, but *cannot* use the `trainable` keyword more than a single time. Any layer definition *can* use the optional 'name' parameter to define a layer name for use in the run section of the graph; and 'name' *cannot* be used as the name of a parameter. Layer variables are not public; only the return of the layer can be accessed.

Optimizers

Onion also defines a subset of layer objects, called optimizers. Declaring an optimizer layer is syntactically identical to normal layers, only we replace the layer keyword with optimizer:

```
optimizer SGD(float loss, float step_size) {...}
```

Every composite model, i.e. a model including all its nested models (if any), must include an optimizer. The optimizer layer implicitly has access to all the **`trainable`** variables used by the other layers in the model. This allows Onion to update trainable variables without the user explicitly passing them as parameters.

Models

Models represent the flow of data from one layer to the next. A model does not define computation; rather, it lays out how computational units (i.e. layers) interact with one another. The model is therefore composed of layers. An example MNIST model may look something like the following:

```
model MNIST([batch] inputs, [batch] labels [, int batch_size = 10]) {
	inputs =>
	FC(64, ‘layer1’) =>
	Relu() =>
	FC(labels.shape[0], ‘layer2’) =>
	predictions =>
	SoftmaxCrossEntLoss(labels) =>
	float loss =>
	SGD(0.5);
	return predictions;
}
```

The **`model`** keyword *must* be followed with a capitalized string name composed only of upper and lowercase alphanumeric characters. The model definition *cannot* index individual matrices and therefore *cannot* do manipulations of individual floats. The model definition *can* use the pipe operator and *can* pipe inputs into layers or variables. Variables *must* be a non-capitalized string name composed only of lowercase alphanumeric characters and underscores. The model definition *can* have a specific optimizer associated with it. Models *can* be nested; if a model is called inside another model, the optimizer closest to that model is used. All layers called in a model *must* be accessible by at least one optimizer. All variables in a model are public and *can* be called in the run scope.

Batching

A batched input is defined as an input for which the first dimension represents a batch size. The optional parameter **`batch_size`** determines whether the model should treat any parameters marked with the **`batch`** keyword as a batched input. If the programmer passes a **`batch_size`** which is larger than the shape of the actual input, we will default to using the entire input. If **`batch_size`** does not equal zero, at least one model parameter *must* have the batch keyword. **`batch_size`** is a reserved parameter name and *cannot* be reused.

Run

The run section of an Onion program is the entry point for program runtime. The run definition is where IO and graph construction occur. The run section therefore acts as the entryway for data to interact models and provides control flow to indicate how the model should be treated. The run section also acts as the exit for data from models. To complete the MNIST example, an MNIST run section is provided below:

```
run(smatrix args) {
      images = FileRead(<path>);
      labels = FileRead(<path>);
      model mnist = MNIST(images, labels);

      train {
            for (int i; i < 1000; i = i + 1) {
                  mnist(images, labels);
                  Print('Step: ', i);
                  Print('Loss: ', mnist.loss);
            }
      }

      predictions = mnist(images, labels);
      Print('Predictions: ', predictions);
}
```

The **run** keyword *must* take in a smatrix args representing any command line arguments passed into the Onion program. The run definition *can* do file IO using the FileRead and FileWrite functions. The run definition *can* initialize graphs using model initialization syntax. The run definition *can* use the train {} keyword. The run definition *cannot* call individual layers, index matrices, or use the pipe operator.

Graph Creation

When a model is initialized, it passes in the data that it is expecting to use during the actual model run. The data is discarded, but the relevant (i.e. accounting for batch_size) shape is taken for an input to a model. This shape is then used to construct all relevant components of the graph on a preliminary run through. Thus, though graph shapes are not defined at compile time, they are necessarily defined before the user can make use of the model.