High Level Goal/Approach

We want to be able to solve a Machine Learning problem. We are given a "Shape Program" that defines the shape of a computation, and a set of training data (a set of input-output pairs). The Shape Program declares "weight" variables, and defines a "loss function". The values of the weights are unknown to the user -- our goal is to compute them -- and the loss function is a measure of closeness between an expected output and the output generated from a certain set of weight values. Our goal is to compute the unique set of weight values that minimize the loss function (we use the Gradient Descent Algorithm, more details are below). We can then use these loss-minimizing weight values to execute the given computation on an arbitrary input outside the training data set. To achieve this, we follow a three-phase architecture.

Three-Phase Architecture:

- *Compile Phase (Shape Program → Gradient Computing Program)*
    - The user provides a Shape Program that defines the shape of a computation. The Shape Program declares inputs, expected outputs, observed outputs, weights, intermediate variables, and a loss function, and defines the mathematical relationships between these variables. The Shape program is to be written in a language called TenFlang (TENsor Flow LANGuage) that we define. In a TenFlang program, variables are either defined as constants, or as simple operations of previously defined variables. The Shape Program must be a *straight-line program*, without any control flow statements.
    - The compile phase uses the Backpropagation Algorithm to generate a program called the Gradient Computing Program (GCP), also in TenFlang. The GCP takes in a fixed weight vector, and computes the gradient of the loss function for a given input vector and a given weight vector.

- *Weight Calculation Phase (GCP + Training Data → Loss-Minimizing weight values)*
    - Using the Gradient Descent method, we determine the weight vector that minimizes the loss function for the given training data. Together, the weight vector and the Shape Program uniquely define a computation. (The GCP created by the Compile Phase is crucial to the Weight Calculation Phase. We write a TenFlang interpreter, which we use to run the GCP on various combination of weight vectors and training data pairs, till we find the loss-minimizing weight vector.)

- *Evaluation Phase (Shape Program + Weights + Input → Output)*
    - Given the Shape Program, an input vector, and a weight vector, we evaluate the output and loss. We use the same TenFlang interpreter to run the Shape Program on the given input and weight vectors.

*** Note: This document describes the high-level concepts behind the system. For finer implementation details or documentation, see the Guide to TenFlang or the API Guide.

Before delving into a detailed description of the TenFlang language or the three phases, this is a high-level overview of how a user would use our programs:

```
$ tfcompiler  shape.tf  >  gcp.tf
$ tfweightcalc  gcp.tf  training_data.txt  >  weights.txt
$ tfevaluate  shape.tf  new_inputs.txt  weights.txt
```

Note On Vector Representation

We will frequently be referring to vectors; they might be vectors of weights, inputs, or expected outputs. We establish the following representation of vectors. Vectors are to be represented as an ordered dictionary of {variable name, value} pairs. The order allows us to perform entire vector operations like scaling a vector, or adding two vectors. Keeping track of the variable names (using a dictionary instead of an array) makes it easy to operate on individual components of the vector.

For example, a weight vector w may look like $\{w_0 : 3, w_1 : 5, w_2 : -10\}$.
An input vector x may look like $\{x_0 : 17, x_1 : -23, x_2 : 0\}$.

Hereafter, when we refer to a vector, we mean this type of name-value dictionary.

Recurring Example We Will Use

Over the next several sections, we will describe the TenFlang language and the Three-Phase Protocol in the context of a simple example computation. This example uses only the simplest features of TenFlang, and only has one input, one output, and one weight; there are no vectors or advanced TenFlang features, but this simple example is enough to illustrate all the principles of our system.

Consider the following computation, described in mathematical terms:

> Let x be an input, w a weight, and y an expected output.
> Let o be the observed output, and λ the loss function variable.

$$o = \frac{1}{1 + e^{(w * x)}}$$
$$\lambda = (o - y)^2$$

We will first learn how to express this computation as a Shape Program in TenFlang. We will next see how the Compile Phase translates the Shape Program into its associated Gradient Computing Program. Then, we will learn about the TenFlang interpreter, and how the Weight Calculation Phase uses this interpreter (along with the Training Data) to compute the value of the weight w. Finally, we will see how the same interpreter is used to evaluate the output of the computation for a given arbitrary input, using the loss-minimizing value of w.

Tensor Flow Language (TenFlang)

Before showing how to express our example computation as a TenFlang Shape Program, we first lay out some basic guidelines for the TenFlang Language. It may be helpful to refer to the example Shape Program (shown below the Guidelines) as you read, to better understand the guidelines.

TenFlang Guidelines

- There are six types of variables:
  - *input*
    - An input to the computation.
  - *exp_output*
    - The expected output for a given input. Expected outputs are typically involved in calculating the loss function.
  - *weight*
    - A free variable, the value of which is not initially known to the user. The ultimate goal of this system is to compute the weights.
  - *intvar (intermediate variable)*
    - An intermediate variable in the computation.
  - *output*
    - The observed output of a computation. Outputs are a function of inputs and weights.
  - *loss*
    - An expression that measures the deviation of the output from the expected output. Loss variables are typically a function of outputs and expected outputs.

- Every line is either a variable declaration or a variable definition:
  - A variable declaration declares the type of a variable, much like type declarations in Java or C++. Lines 1-7 are examples of variable declarations:
  - A variable definition defines the value of a variable as an expression involving other variables and/or constants. Lines 9, 11 and 12 define variables using binary operations, and Line 10 defines a variable using a unary operation.
  - Every variable must be declared before it is defined.

- Binary operations include addition, multiplication and exponentiation. Unary operations include the logistic operation, natural log, and base-e exponentiation. It is critical that these operations be differentiable, in order for the Gradient Descent Algorithm to work (Weight Calculation Phase).

Basic Syntax Guidelines

- Every token must be separated by a space.
- Declarations must be of the format:
  - `declare <type> <variable_name>`
- Definitions must be in one of the following formats:
  - `define <variable_name> = <operator> <operand> <optional second operand>`
  - `define <variable_name> = <another variable name>`
  - `define <variable_name> = <constant>`

*** Note: There are many more advanced features of TenFlang (vector operations, macros, etc.) that are not mentioned in this document. See the Guide to TenFlang for more details.

Example TenFlang Shape Program

Computation: $o = \frac{1}{1+e^{(w*x)}}$, $\lambda = (o - y)^2$

`example_shape.tf`

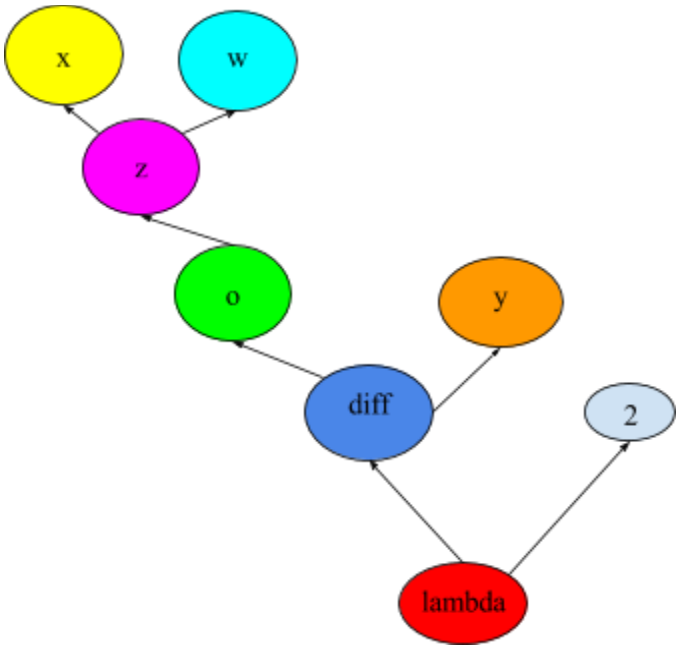
```
1     input  x
2     exp_output  y
3     weight  w
4     intvar z
5     output  o
6     intvar  diff
7     loss  lambda
8
9     z  =  mul  x  w
10    o  =  logistic  z
11    diff  =  sub  o  y
12    lambda  =  pow  diff  2
```

Compile Phase

First, we describe how the Compile Phase works, and follow along with our example computation. The goal of the Compile Phase is to generate a Gradient Computing Program, which computes the partial derivatives of the loss function with respect to each of the weights. Below, the resulting example Gradient Computing Program is shown. It may help to refer to the example GCP in parallel, to better understand how the Compile Phase works.

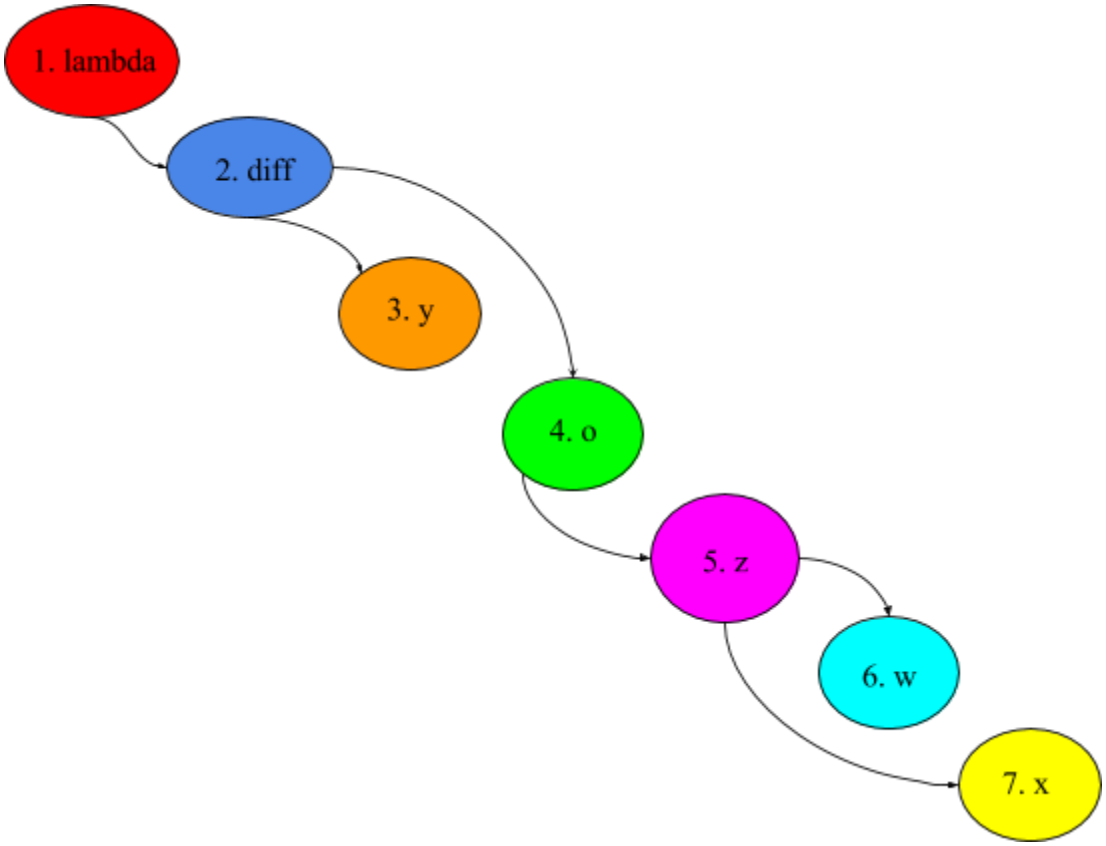
How the Compile Phase Works

- *First, parse the Shape Program, and create a Data Flow Graph from it.*
    - Each variable in the Shape Program is represented by a node. Input, expected output, and weight nodes have no children; they take no inputs (they are given a value at interpretation time), and emit an output, namely, their value. Other nodes (intermediate variables, outputs, loss variables) may have one or two children; they take one or two inputs and emit one output. These nodes represent operations on their inputs.
    - For every node *v* with feeder nodes (intvars, outputs, loss variables), there is a directed edge from *v* to its feeders (children). The direction of these edges may seem counter-intuitive now, but this scheme allows us to think of the Data Flow Graph as a dependency graph for generating our GCP. If the outputs of nodes A and B are inputs to node C, the partial derivatives with respect to A or B are dependent on the partial derivatives with respect to C. We will explore this soon.



Data Flow Graph for our example Shape Program

- *Next, topologically sort our Data Flow Graph*
    - To understand why we topologically sort our Data Flow Graph, consider one of our main objectives: computing the gradient vector of the loss function; that is, computing the partial derivatives of the loss function with respect to each one of our weights w_i.
    - In our example, our loss function λ is a function of the intermediate variable *diff*. We ultimately need to compute $\partial\lambda/\partial w$ (since there is only one weight, we need only compute one partial derivative). However, by the Chain Rule, $\partial\lambda/\partial w = \partial\lambda/\partial z * \partial z/\partial w$. In order to compute $\partial\lambda/\partial z$, we'd first need to compute $\partial\lambda/\partial o$, and so on. A topological sort of the Data Flow Graph gives us an order in which to go about evaluating these partial derivatives in the GCP.



Topologically Sorted Order of Nodes

- *Finally, generate the code for the Gradient Computing Program (GCP)*
    - The first step in generating the GCP is to copy (with a few variations) the lines of the Shape Program (Lines 1 through 7 below). The only changes we make are in the variable declarations:
        - Inputs, expected outputs, and weights from the Shape Program all become inputs in the GCP. Outputs, intermediate variables and loss variables all become intermediate variables in the GCP.
        - The definitions of these variables are identical. It is important we define these variables. Since our partial derivatives are often functions of our variables, we must compute the values of these variables before we can begin computing partials.
    - Now, we can begin thinking about partial derivatives. We iterate over the nodes of our Data Flow Graph, in the topologically sorted order. Remember, each node represents a variable in the Shape Program. For each variable v in our traversal, we do the following:
        - Declare $\partial\lambda/\partial v$ as an intermediate variable.
            - If v is a weight variable in the Shape Program, instead declare $\partial\lambda/\partial v$ as an output variable (Line 25). After all, the partial derivatives of the loss with respect to the weights are the desired outputs of the Gradient Computing Program.
        - If v receives an edge from its "parent node" p in the Data Flow Graph (if the output of node v flows to node p), then use the Chain Rule and define $\partial\lambda/\partial v$ as $\partial\lambda/\partial p * \partial p/\partial v$ (Lines 10, 13, 16, 18, 21, 25, 27). Note that both $\partial\lambda/\partial p$ and $\partial p/\partial v$ will have been defined in a previous iteration, since p is necessarily before v in the topological sort. (Perhaps this clarifies how we've chosen to direct the edges in our Data Flow Graph.)
        - If v is a binary node, declare intermediate variables $\partial v/\partial m$ and $\partial v/\partial n$ where m and n are the two "children nodes" of v (The outputs of nodes m and n flow to node v). If v is a unary node, declare the intermediate variable $\partial v/\partial m$, where m is the lone child of v.
        - Define $\partial v/\partial m$ and/or $\partial v/\partial n$ using simple differentiation. Recall that v is a simple differentiable binary function of m and n (Lines 11, 13, 14, 19, 22, 23).

A Quick Note on Runtime:

At each node v we visit in topologically sorted order, we only add a small constant number of lines (the declarations and definitions of $\partial\lambda/\partial v$, $\partial v/\partial m$, and $\partial v/\partial m$. This means that the time it takes to interpret (run) the GCP is proportional only to the number of variables in the the Shape Program, rather than the product of the number of weights and the number of variables.

The Backpropagation Algorithm we are using here effectively allows us to reuse values that we've already computed. For example, $\partial\lambda/\partial z = \partial\lambda/\partial diff * \partial diff/\partial o * \partial o/\partial z$ and $\partial\lambda/\partial o = \partial\lambda/\partial diff * \partial diff/\partial o$. Both $\partial\lambda/\partial z$ and $\partial\lambda/\partial o$ depend on $\partial\lambda/\partial diff$. However, observe that we only compute the value of $\partial\lambda/\partial diff$ once (Line 11). When we interpret (run) our GCP, all subsequent references to $\partial\lambda/\partial diff$ simply use the value computed in Line 11. This saves a lot of computation in the Weight Calculation Phase.

Example TenFlang Gradient Computing Program (GCP)

For ease of understanding, we will write the GCP in pseudocode/mathematical terms, instead of in exact TenFlang syntax. The ideas are exactly the same.

```
1     input x, y, w
2     intvar o, diff, lambda
3
4     z = x * w
5     o = logistic z
6     diff = o - y
7     lambda = diff^2
8
9
10    intvar ∂λ/∂λ = 1
11    intvar ∂λ/∂diff = 2 * diff
12
13    intvar ∂diff/∂o = 1
14    intvar ∂diff/∂y = -1
15
16    intvar ∂λ/∂y = ∂λ/∂diff * ∂diff/∂y
17
18    intvar ∂λ/∂o = ∂λ/∂diff * ∂diff/∂o
19    intvar ∂o/∂z = logistic_deriv z
20
21    intvar ∂λ/∂z = ∂λ/∂o * ∂o/∂z
22    intvar ∂z/∂x = w
23    intvar ∂z/∂w = x
24
25    output ∂λ/∂w = ∂λ/∂o * ∂o/∂w
26
27    intvar ∂λ/∂x = ∂λ/∂o * ∂o/∂x
```

Weight Calculation Phase

First, we describe how our TenFlang interpreter works, and then describe how our interpreter plays a role in executing the Gradient Descent Algorithm. In the Gradient Descent Algorithm, we use our interpreter to repeatedly run our GCP on given weight and input values. This iterative update algorithm terminates once we've found the values of the weights for which the loss function (defined in the Shape Program) is minimized.

How the Interpreter Works

First, we recall the inputs to the interpreter. The interpreter takes in a program, and a vector of input variables. The input vector given to the interpreter is structured as a set of {variable name, value pairs} (Recall our representation of vectors). For example, the input set may look like $\{a\colon 3, b\colon -4, c\colon 0\}$. We will refer to this set of inputs as the Input Dictionary. The values in the Input Dictionaries will be substituted in place of the corresponding *input* and *weight* variables in the program.

The main data structure for the interpreter is a dictionary of {variable name, value} pairs. This dictionary, call it the Bindings Dictionary, contains the name-value bindings for every variable in the program. The Bindings Dictionary is built up over the course of interpretation. After the entire program is read, the interpreter looks up the values of all the *output* variables in the Bindings Dictionary, and returns these values in a vector.

In order to build up the Bindings Dictionary, the interpreter reads one line of the program at a time, and performs an action, based on the contents of the line. (Recall every line is either a variable declaration or a variable definition.) For every line, the interpreter will do one of three things:

- If the line is a declaration of an *input* or a *weight* variable, a name-value binding will be added to the Bindings Dictionary. The name will be the name of the variable, and the value will be the appropriate value from the Input Dictionary.
- If the line is a declaration of any other type of variable (*exp_output, output, intvar or loss)*, a dummy name-value binding will be added to the Bindings Dictionary. The name will be the name of the variable, and the value will be NULL.
- If the line is a definition of a variable v, we evaluate the expression for the variable using existing bindings, then set the value of v in the Bindings Dictionary.
    - For example, if the line is `v = add a b,` we perform two lookups in the Bindings Dictionary to get the values of *a* and *b*, add them, and set the result to be the value of *v* (which was previously NULL) to be $a + b$.
    - If either of the operands *a* or *b* is NULL, the resulting value of *v* will be NULL, as well.

We walk through the interpretation of a very simple TenFlang program, to better understand how the interpreter works. Note that this program is a valid TenFlang program, but does not necessarily resemble either a Shape Program or a Gradient Computing Program. This is done intentionally, to emphasize that **the interpreter works on *every* valid TenFlang program**. This allows us to use it to interpret a GCP (shown later in this phase), or to interpret a Shape Program (shown in the Evaluation Phase).

Initially, the Bindings Dictionary is empty. We note the updated version of the Bindings Dictionary next to every line. (`--` is used in place of NULL.)

```
Input Vector: {x: 3,  w: -2}

1     input  x         {x: 3}
2     weight w         {x: 3, w: -2}
3     exp_output y     {x1: 3, w: -2, y: --}
4     output o         {x1: 3, w: -2, y: --, o: --}
5
6     intvar a         {x1: 3, w: -2, y: --, o: --, a: --}
7     intvar b         {x1: 3, w: -2, y: --, o: --, a: --, b: --}
8     intvar c         {x1: 3, w: -2, y: --, o: --, a: --, b: --, c:
--}
9
10    c = mul y w      {x1: 3, w: -2, y: --, o: --, a: --, b: --, c:
--}
11    a = mul x w      {x1: 3, w: -2, y: --, o: --, a: -6, b: --, c:
--}
12    o = add a w      {x1: 3, w: -2, y: --, o: -8, a: -6, b: --, c:
--}

Return {o: -8}
```

Note a couple of things:

- The *exp_output* variable *y* was declared in Line 3, but never got defined. This is not a problem.
- When the *intvar* variable *c* was defined (Line 10), one of its operands, *y,* was NULL, so *c*'s value got set to NULL.

Gradient Descent Algorithm

The objective of the Gradient Descent Algorithm is to find the weight vector that minimizes the value of the loss function. The loss function for a particular {*input*, *exp_output*} pair is defined in the Shape Program. It is typically a function of the inputs, weights and expected outputs. We will seek to find the weight vector that minimize the average of the loss function over all the pairs in the Training Data.

The algorithm is described in pseudocode, and works as follows. Recall that the `interpret` method called in `find_partials` is the TenFlang interpreter we described above!

```
SUBROUTINE
find_partials(GCP, weight_vector, pair):
      inputs = union(weight_vector, pair)
      return interpret(GCP, inputs)

SUBROUTINE
avg_gradient(GCP, weight_vector, training_data):
      sum_of_partials = vector_of_zeros()
      for all {input, exp_output} pairs in training_data:
            sum_of_partials += find_partials(GCP, weight_vector, pair)
      return component_wise_div(sum_of_partials, len(training_data))

MAIN ROUTINE
calculate_weights(GCP, training_data):
      weight_vec = initial_guess()
      grad = avg_gradient(GCP, weight_vec, training_data)
      while !(grad ≈ 0):
            increment_vec(weight_vec, −δ*grad)
            grad = avg_gradient(GCP, weight_vec, training_data)
      return weight_vec
```

We summarize the salient points:

- We start with an initial guess for the weight vector.
- We calculate the average gradient of the loss function, given our guess. The average is over all the pairs in the Training Data. This is where we use our interpreter.
- We push our guess for the weight vector in the opposite direction of the loss function's gradient (`increment_vec(weight_vec, −δ*grad)`), and repeat until the gradient is zero (or close enough).
- Upon termination, we return the vector of weights that minimizes the loss.

Evaluation Phase

The evaluation phase is the simplest of the three. We take the Shape Program we've had all along, and the weight vector w^* outputted by the Weight Calculation Phase. Additionally, we are given a new input vector x^*, NOT in the training data. Our task is to run the Shape Program with the weights w^* and the input x^*. Recall that to "run" a program is simply to pass the program and its inputs to our interpreter.

We briefly describe the Evaluation Phase using pseudocode:

```
MAIN ROUTINE
evaluate_output(Shape Program, weight_vector, input_vector)
      inputs_to_shape_program = union(weight_vector, input_vector)
      return interpret(Shape Program, inputs_to_shape_program)
```