Andrei Spiride
James Adams
Felipe Morales Osorio

Final Project Report

Overview

Our project was to build an extensible deep learning system within Scheme. Taking some influence from the work of Andrej Karpathy and George Hotz in developing micrograd (https://github.com/karpathy/micrograd) and tinygrad (https://github.com/geohot/tinygrad) respectively and building much of our system, as suggested by the professor, off of 'The Little Learner' by Daniel P. Friedman and Anurag Mendhekar, our goal was for our project to be able to build and train an assortment of neural networks that the user can input themselves. Additionally, a system from scratch was implemented as well.

Project Repository

Link to Repository:

https://github.com/jkaohu01/6.5151-Final-Project

Repository Structure:

Let <root> denote the root directory of the repository. The files for the system from scratch are stored in the directory: <root>/scheme-pytorch/. The files for the Little Learner system are stored in the directory: <root>/sdf/deep-learning/. To load all the files required for the Little Learner system, first run (load "<root>/sdf/manageryhuihuy/load") to load the sdf file manager and then run (manage 'new 'deep-learning) to load the load-spec required for deep learning. The sdf files were only used to run the loader but nothing else was used for the Little Learner system.

Test Cases:

Every file in the <root>/sdf/deep-learning/ directory contains numerous test cases for the procedures implemented. These test cases can be found directly below the procedure in a comment block. Feel free to explore these test cases to understand what each procedure accomplishes. Additionally, the most relevant tests for the Little Learner system are found in <root>/sdf/deep-learning/example-network.scm which contains tests for two simple neural networks that can be run. This file contains instructions on how to run the code in commented

blocks. The two test cases are a simple neural net trained on only one data point which will run relatively fast. There is also a much larger neural net that trains on XOR data to learn this function. However, this will take much longer to run.

Implementing Little Learner System

Following along with "The Little Learner" proved to be more difficult than originally anticipated. One main issue we originally encountered was that the book starts off by explaining that the malt package is required to run the programs implemented in the book. This package is available for free download on the book website but it is written in Racket so we first had to translate it into MIT-Scheme. At first, we attempted to translate the code from the github repository into MIT-Scheme but this proved to be too difficult without explanation of what the code was supposed to do. Part of the difficulty of this translation involved the fact that none of us knew Racket so it was difficult to determine which functions were native to Racket and which were implemented in the package.

Eventually, we found that much of the code in the repository is covered in Appendix A and B of the book. These appendices cover the implementation of duals, tensors, the gradient function, and the definition of primitive operators that work on duals. However, other portions of the malt package are scattered throughout the text. In particular, the extension of operators compatible with tensors is covered in Interlude V of the book. Below we will discuss how we implemented these functions from the book as well as additional code that was simultaneously developed in case we could not get the functionality we desired from the textbook.

Explanation of Little Learner System

Gradients:

To update the weights of a neural network, we need to know how to calculate the gradient of a loss function. The resulting gradient tells us how to update the weights of a neural network to reduce the loss. This is usually accomplished by backpropagation which requires some form of automatic differentiation. The backpropagation will walk backwards through some tree of operations and operands that produced the loss until it reaches some input value to the loss function. When finished, we will know how much influence each input had on the output of the

function (in terms of the partial derivatives of the output with respect to the inputs). This is the gradient.

The key idea for backpropagation is accumulating how much local influence each component of the tree had on the output while walking backwards. If we combine all the local influences, when we reach an input value we will know how much total influence the input value had on the output. In other words, first, we need to find some local gradients based on the parent operands and parent operator that produced the current value in the tree. The local gradients express the partial derivative of the current value with respect to each parent operand or, equivalently, each parent value. Next, if the current value is itself an operand producing some child value, we should multiplicatively accumulate each one of the local gradients with the accumulated gradient value received from the child value. Finally, we can transfer each of these multiplicatively accumulated results to their respective parent value in the tree. Each one of the transferred results expresses the influence that parent had on the output or, equivalently, the partial derivative of the output with respect to the parent. This is simply an application of the chain rule from calculus.

To do the method described above, we need some way of storing the history of operands and operators that produced each value as well as a local gradient function that tells us how to multiplicatively accumulate, or propagate backwards, the local gradients. Duals are the data structure used by the book to accumulate local gradients until we reach some input to the function we are taking the gradient of. When we reach an input value, we can simply associate the multiplicatively accumulated gradient with the input value in some table.

Gradient state-table:

This is the table that associates every input to the partial derivative of the output with respect to that input. Initially, it is empty but it will be filled up as we backpropagate as described. The full table contains all the partial derivatives required for the gradient of a function.

Duals:

Scalars in the book are represented as duals. Duals are composed of two pieces. The first piece is a numeric value representing the result of some operation or a numeric input value that

has no prior values that created it. The numeric value is referred to as the *real-component*. The second piece of a dual is the link function which is what does the backpropagation. The link function is referred to as the *link-component*.

Truncated Dual:

This is simply a dual whose *link-component* is the special end-of-chain procedure. Links call other links so the end-of-chain link is what ends the call of links. Truncated duals are used to represent the input to a function we are taking the gradient of.

end-of-chain link:

This is the link that ends a chain of link calls. It also updates a gradient state-table because whenever one is called it means we have reached an input to some function we are taking the gradient of. Since we reached an input, we can associate, in the state-table, the input with the multiplicatively accumulated gradient. This associates the input with the partial derivative of the output with respect to the input.

Link Functions:

Link functions take in three arguments. The first is the dual associated with this link. This does not strictly have to be the case. We can pass any dual as the first argument and no errors will be thrown. However, for backpropagation to work, we need a reference to the dual whose link we are calling when we are in the body of the link so we must follow the convention of passing the dual associated with this link as the first argument. Call this value *self-dual*.

The second argument to a link is a multiplicatively accumulated gradient received from a child dual. Again, this is convention. Call this value *accumulator*. When a dual has no child, we simply pass the multiplicative identity, 1.0, as the argument.

The third argument to a link is a gradient state-table that is passed down but only updated by the end-of-chain link. All other links must do nothing with this state-table. The table is where we eventually will store an *accumulator* received from a child dual. Call this argument *state-table*.

Links have access to an environment containing the parent operand duals that produced *self-dual* as well as a procedure for finding the local gradients based on the operator that the

parent duals were combined with. Note that for this to occur automatically, we have to define every operator we use in a special way which will be covered later in this document.

We can use the local gradient procedure, the *accumulator* provided by a child dual, and the reference to the parent duals to implement the backpropagation procedure as described earlier. When a link is called, it finds the local gradient with respect to each parent, multiplicatively accumulates the local gradient with the *accumulator* received from a child dual, and then passes down the accumulated result to the respective parent. The way the result is passed down is by calling the link function of each of the parent duals with the accumulated result as the *accumulator* argument. The *state-table* argument is also passed down to each parent link called. Recall that this will eventually be updated by some input duals but meanwhile all the links leading to those duals do nothing with this table.

Primitive Procedure on Duals:

Primitive procedures are the procedures we implement in a special way so that the functionality of links works. We define a primitive procedure as simply a procedure that takes in two duals as input, or one dual in the simple case that will not be covered since it is trivial to implement once the more complex case is done, and produces one dual as output. The output dual has its *real-component* produced by combining the *real-component*'s of the input duals using some *numeric-function* which outputs a number. The *link-component* of the output dual is created in an environment containing the parent duals so that they can be referenced in the link body. The *link-component* is also created in an environment containing a binding to a *gradient-function* that should be defined as the partial derivative of the output with respect to the two inputs assuming the *numeric-function* is the function combining the two inputs. We have to ensure that the *gradient-function* is correct analytically because we define it manually.

Tensors:

Tensors in the book are the main pieces that build up a neural network. They are represented as vectors, whose elements can either be other tensors or a scalar (dual), or just a single dual. A tensor essentially builds up all the nodes within a neural network and allows the usage of scalar, vector and matrix operations on the tensor to allow for training of the neural network.

A tensor keeps track of two important properties which are its rank and shape. A tensor's rank is just how deep it is or how many layers of embedded tensors there are. The book also simply identifies the rank as the number of left brackets before the leftmost scalar. Shape is a list where the *i*th element describes *i*th dimension of the tensor. For example a tensor such as #(#(4 5) #(3 2) #(0 5) #(9 1)) has a shape of '(4 2) meaning the tensor has four tensors each with 2 scalars. A shape of '(3 3 1) would be a tensor with three tensors each that have three tensors themselves and each of those tensors have one scalar. Shape will be important later on for the reshape procedure that can change a tensors shape, which has not been implemented yet. Reshape would be useful for potentially rearranging a neural networks layout.

Extending Operators to support Tensors:

Primitive operators like +, -, *, / which operate on duals, in the special way we have described before which allows us to automatically differentiate, should be able to operate on tensors as well. The important thing to notice is that we want these operators to act on some base case of tensors. For example, (+ 1 #(1 2)) and (+ 1 #(#(1 2) #(3 4))) should produce #(2 3) and #(#(2 3) #(4 5)) respectively. This is because the + 1 should operate on the rank 0 tensors, which are simply scalars, of the respective tensors. Similarly, (+ #(1 2) #(1 2)) and (+ #(1 2) #(#(1 2) #(3 4))) should produce #(2 4) and #(#(2 4) #(4 6)) respectively because the + #(1 2) will operate on the rank 1 tensors of the respective tensors.

We can redefine the operators to perform its operation on some tensors of some base rank. Then we descend into a tensor until we reach the base rank and apply the operation. To do this, we use the procedure that finds the rank of the current tensor.

Neurons:

Before getting to the actual construction of a neural network we needed to define neurons for our system. A neuron is an individual node within our neural network which needs a nonlinearity function. For both the book and our system the only one we used was relu. We define rectify-0:

```
(define (rectify-0 scalar)
  (cond
   ((<-0-0 scalar 0.0) 0.0)
```

```
    (else scalar)))
```

Which takes a scalar and return 0 if it's negative otherwise it returns the scalar. Then we define rectify which just applies rectify-0 to all values in a tensor. We also need a linear portion for the neuron so we define linear:

```
(define linear
  (lambda (t)
    (lambda (theta)
      (+ (dot-product-2-1 (ref theta 0) t) (ref theta 1)))))
```

This takes a tensor and a set of weights and biases and applies them to the tensor using the dot product acting on tensors of rank 2 and tensors of rank 1 (this is actually equivalent to standard matrix-vector multiplication). Combining linear and rectify we can define a relu neuron:

```
(define relu
  (lambda (t)
    (lambda (theta)
      (rectify ((linear t) theta)))))
```

With this relu neuron we are able to start building our actual networks.

Building Neural Networks:

Tensors are the basic data type used in our networks however more is needed to be defined to build a working and trainable neural network. We define a block in scheme:

```
(define block
  (lambda (fn shape-lst)
    (list fn shape-lst)))
```

As seen here a block is a list of two values. The first value being a target function that we will use to train our network and the second is a list of tensor shapes that will be used to build our tensor thetas. Theta is just a list of tensors, these tensors will represent the weights and biases of our neural network. Block can essentially be thought of as a layer in our neural network, but we will see later that a block can be expanded beyond that. Defining a layer requires both a function and a list of shapes and within the book and our system we only used the relu function when constructing a layer. For example to construct a layer:

```
(define layer1
  (block relu
```

```
(list
 (list 64 32)
 (list 64))))
```

Important note is that the first list within the input list is the shape of the weights, the second being the weight of the biases. If a weight for a layer has the shape (n m) then the biases must be of shape (n) that also means the input size for this layer is (m) and the output size is (n). We then took this definition of a layer and condensed it into the procedure dense-block that automatically uses relu and only needs the input size and output size of the layer.

A network will likely need to be more than one layer, so the next step is a procedure that can combine blocks. We define the function stack-blocks which accomplishes this, it has a number of helper functions, the first being stacked-blocks which is just recursively called for all blocks called in stack-blocks. Then in stacked-blocks the function stack2 is called which takes two blocks and makes a new block from those two. Finally we define the procedure block-compose which generates a new target-function from the two block target functions that are inputted. All together stack-blocks are then able to take in any number of layers/blocks and combine them into one neural network, or essentially a larger block. To define a network is then very simple for example:

```
(define network
  (stack-blocks
   (list
    (dense-block 4 6)
    (dense-block 6 3))))
```

Important note here is that the input size of a layer must match the output size of the previous layer, as we see in this example the first layer's output size is 6 so the second layer's input size must be 6.

With this network we have what we need for gradient descent. Specifically we need an objective function, theta and the data we are gonna train on. The object function is our loss function taking in our target function as an input, and the target function is very easily extracted from our network because it is still a block so it's still a list of a target function and a list of shapes. So the objective function is very easy to get from a network, the theta is a little harder to get because our second value in the block is just a list of shapes not a theta. Now one possibility

is to set the theta tensors to all be 0.0, but this wouldn't be an effective network as every weight and node would not learn anything different. So instead we created a procedure that will randomize the weights to a value between -1 and 1, and the biases all set to 0.0:

```
(define init-shape
  (lambda (s)
    (cond
     ((= (len s) 1) (zero-tensor s))
     ((= (len s) 2)
      (random-tensor s)))))
```

Zero-tensor creates a tensor of all 0.0, used for the biases. Random-tensor creates a tensor with all values between -1 and 1. So this allows all the nodes to not do the same training which would be redundant. Now with this randomized theta and target function, gradient descent and the training of our network becomes possible.

We build some example neural networks in example-network.scm that are already to train on basic functions for the most part. One issue we found was the randomization of theta would sometimes lead to a poor network that did not converge. Overall our little learner system was able to accomplish some basic machine learning and if we were to extend on this system we would include optimization and more nonlinearity functions such as sigmoid and tanh.

Implementation from Scratch

In parallel to implementing the code from the book, we wanted to see if independently, we would come up with similar structures/solutions working through the problem of building a Pytorch counterpart. Essentially, we wanted to see if the same ideas for structure came to us when we thought about solving this problem.

This implementation succeeded in providing the infrastructure to easily build, train, and test networks. It supports many similar features such as hyperparameters, automatic differentiation/backpropagation, batching, and inputs of any size. However, we noticed 2 key differences in the structure of implementation.

The first difference is in the root implementation of the neuron. In the implementation from scratch, we determined the simplest way of representing a neuron was as a record. This way, its contents, containing info such as its value, gradient, children, backward procedure, and the operation that was performed to create this neuron, could easily be accessed. The book took a more functional approach. The functional approach makes it more difficult to check the behavior at every individual neuron. We believe that using the records structure allows you to more easily inspect any neuron and understand what's going on at any point in the process. Everything is built up from this neuron in our implementation. A layer contains many neurons and a network contains many layers.

The second difference is in how we keep track of a node's gradient in relation to any loss. In the textbook, they store this as the link, where every node stores the path to get to the loss neuron. This is different from the approach we came up with independently. We decided to make every neuron point to its children. This creates our graph. In order to get the ordering or update of all the gradients for all our weights, we simply run a topological sort on our graph, and then update the nodes in that order. This makes sure we can always do the chain rule in the correct order. We think this is likely more efficient in terms of memory. For the implementation from the book, as networks become bigger, storing entire paths with every neuron as a link could be too memory hungry.

Overall, we were not surprised to see our two implementations were relatively similar. The structure we built on top of the base neurons in both flavors of Pytorch were pretty much identical and they both worked as expected. One general problem we ran into was memory use of Scheme as a whole. We weren't sure how to approach the problem of reading a lot of data at

once in Scheme. As expected, this is necessary when dealing with neural networks. Typically, you aim to train large networks on thousands, if not millions of data points.

Using our implementation from scratch, we tried to tackle the MNIST dataset. We downscaled the image sizes to 10x10 pixels, and only tried to identify 0s and 1s. Using a training dataset of 10,000 samples did not run to completion unfortunately. We encountered memory issues when trying to train in the REPL. What is needed here is probably a more efficient way of training in which you load small batches of data into memory, one at a time. An example of doing this might be to read files directly using Scheme, instead of reading files in python and saving the data as a list in Scheme. We didn't have enough time to play around with this data loading structure. Trying to parallelize these networks would also be a cool problem to tackle. There are cool applications of online and offline learning in the domain of ML.