C?

Andrew Aday, Amol Kapoor, Jonathan Zhang

Overview

- Background
- Implementation
  - Syntax
  - Program Structure
  - Features
- Libraries
  - Math
  - DEEP
- Demo

Design Goals

- Languages are made or broken by their libraries
  - Python: Numpy, Pandas, Theano, Tensorflow
  - Ruby: Rails
  - Prolog: …?

- What does a library need?
  - Easy to use, hard to break: strong typing, yet familiar syntax
  - Custom types for extensibility: structs
  - Abstracting calls from definitions: function pointers
  - Heavy data crunching: matrices
  - Links to other languages with better libraries

Implementation: Syntax

Basically C

- {} for scoping
- Lines end with ;
- Variables declared as `typ NAME`
- Requires `int main() {}` as execution entry point

There’s some Go. Andrew wanted it.

```
int main() {
    int i;

    i = 0;
    while ( i < 10 ) {
        i = i + 1;
        print( i );
    }

    for (i = 0; i < 10; i = i + 1) {
        print( i );
    }
    return 0;
}
```

Implementation: Program Structure

- Statically Scoped
- Declarations for structs/functions/variables must come before use
- Standard Control Flow
  - If...else…
  - While, For
  - Return
- Didn’t stray from MicroC - was not our area of interest

Features: Arrays

- Every array has 8 bytes overhead
  - Total size in bytes
  - Length
- Array literals
- Dynamic array resizing
- Concatenation and Append

```
int[] a;
int[] b;

a = (int[]) {1,2,3};
b = (int[]) {4,5,6};

a = concat(a,b); // {1,2,3,4,5,6}
a = append(a,7); // {1,2,3,4,5,6,7}
print(len(a));   // 7
```

Features: Structs

- Arbitrary collection of custom types
  - Nested structs
  - Arrays

- Method Dispatch

- Allocated on Heap, pass by reference

```
struct rectangle {
  float: width, length;
}

[struct rectangle r] area() float {
  return r.width * r.length;
}

int main() {
  struct rectangle rectangle;

  rectangle = make(struct rectangle);  // malloc space
  rectangle.width = 3.0;
  rectangle.length = 4.0;
  print_float(rectangle.area());       // 12.0
}
```

Features: Function Pointers

- Abstract function calling from function definition

- Allow for creation of modular plug and play components

```
int add(int x, int y) {
    return x + y;
}

int mult(int x, int y) {
    return x * y;
}

/* In the function pointer type below, the last value type is the return */
void print_bin(fp (int, int, int) f, int x, int y) {
    print(f(x, y));
    return;
}

int main() {
    print_bin(add, 7, 35);        /* 42 */
    print_bin(mult, 7, 6);        /* 42 */

    return 0;
}
```

Features: C Links

- Link to any C code with `extern` keyword

- Provide C code in `/lib/` folder

- Compiler combines C LLVM with generated LLVM for single executable

```
extern void printbig(int c);

int main() {
    printbig(72); /* H */
    return 0;
}
```

Features: Matrices

- Matrix implementation through eigen library

- Large number of eigen operators available, built-in

```
int main(){
    fmatrix fm1;
    fmatrix fm2;
    fmatrix fm3;

    /* Create a 5 by 5 matrix of zeros */
    fm1 = init_fmat_zero(5, 5);
    /* Create a 5 by 5 matrix of 2.5's */
    fm2 = init_fmat_const(2.5, 5, 5);

    /* Matrix literal */
    fm3 = [[1.0, 2.0, 3.0], [4.0, 5.0, 60], [7.0, 8.0, 9.0]];

    print_mat((fm1 + 1.0) + fm2);
    fm1 = fm1 + 1.0;
    print_mat((fm1 + 12.0) .. fm2);    /* Matrix multiplication */
    print_mat(fm1 * fm2);                   /* Hadamard product */

    return 0;
}
```

Libraries: Math

- Goal: Build generic library that uses externed code mixed with self built code

- Implementation:
  - Extended a significant portion of C standard math library, including trig, exp, log functions
  - Built basic number manipulation extensions
    - e.g. max, min
    - e.g. sqrt, square
  - Combined eigen math library with own code to build useful distributions
    - e.g. rand_norm() pulls a random number from an input normal distribution
    - e.g. sigmoid() returns a defined value from the sigmoid distribution

Libraries: DEEP

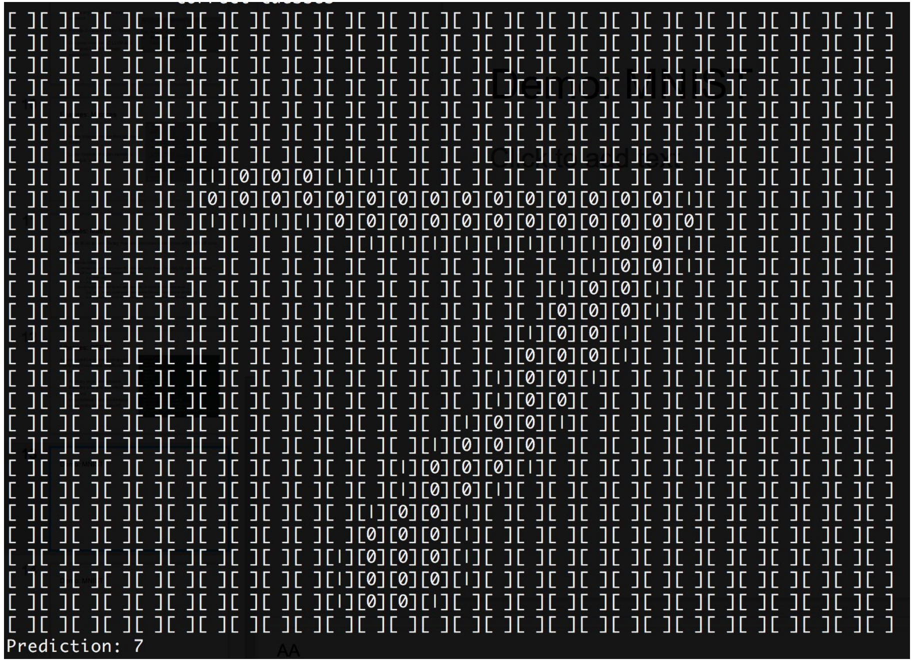
A basic machine learning library for easily fully-connected, feedforward models

- Arbitrary layer architecture
- Arbitrary cost and activation functions
- User-defined hyperparameters
- Uses every single feature!!

```
struct fc_model {
  fmatrix[] train_x;
  fmatrix[] train_y;
  fmatrix[] test_x;
  fmatrix[] test_y;
  fmatrix[] biases;
  fmatrix[] weights;
  int[] layer_sizes;
  int epochs;
  int mini_batch_size;
  float learning_rate;
  fp (float) weight_init;
  fp (float, float) activate;
  fp (float, float) activate_prime;
  fp (fmatrix, fmatrix, float) cost;
  fp (fmatrix, fmatrix, fmatrix, fmatrix) cost_prime;
}
```

Demo: MNIST

- Benchmark machine learning problem
- 28x28 grayscale images of handwritten digits
- 60,000 training
- 10,000 test



Demo: MNIST

- 97.2% classification accuracy

```
Training Epoch 19: [==============================]
        test set cost: 262.099349
        test set accuracy: 9715/10000 = 0.971500
Training Epoch 20: [==============================]
        test set cost: 269.835240
        test set accuracy: 9720/10000 = 0.972000
```

```c
epochs = 20;
learning_rate = .1;
mini_batch_size = 10;
layer_sizes = (int[]) {784, 50, 10};

/* allocate memory */
fc = make(struct fc_model);

/* Popuate fc model fields */
fc.train_x = train_fm_images;
fc.train_y = train_fm_labels;
fc.test_x = test_fm_images;
fc.test_y = test_fm_labels;
fc.layer_sizes = layer_sizes;
fc.epochs = epochs;
fc.mini_batch_size = mini_batch_size;
fc.learning_rate = learning_rate;
fc.weight_init = norm_init;
fc.activate = sigmoid;
fc.activate_prime = sigmoid_prime;
fc.cost = cross_entropy_cost;
fc.cost_prime = cross_entropy_cost_prime;

fc.train();
fc.demo(5);
```