Gravity: An Artificial Neural Network Compiler for Embedded Applications

Tony Givargis
Department of Computer Science
University of California
Irvine, CA 92697-3435
givargis@uci.edu

***Abstract*—This paper introduces the Gravity compiler. Gravity is an open source optimizing Artificial Neural Network (ANN) to ANSI C compiler with two unique design features that make it ideal for use in resource constrained embedded systems: (1) the generated ANSI C code is self-contained and void of any library or platform dependencies and (2) the generated ANSI C code is optimized for maximum performance and minimum memory usage. Moreover, Gravity is constructed as a modern compiler consisting of an intuitive input language, an expressive Intermediate Representation (IR), a mapping to a Fictitious Instruction Set Machine (FISM) and a retargetable backend, making it an ideal research tool for exploring high-performance embedded software strategies in AI and Deep-Learning applications. We validate the efficacy of Gravity by solving the MNIST handwriting digit recognition on an embedded device. We measured a 300x reduction in memory, 2.5x speedup in inference and 33% speedup in training compared to TensorFlow. We also outperformed TVM, by over 2.4x in inference speed.**

Keywords—Artificial Neural Networks, Embedded Software, Compilers for Embedded Systems, Design Automation

I. Introduction

An Artificial Neural Network (ANN) is a numerical information processing system inspired by the way biological nervous systems, including the human brain, manipulate information [1]. Properly configured and trained, ANNs are remarkably good at extracting patterns, extrapolating trends, discovering complex correlations or detecting faint signals in ways that traditional computer algorithms or humans fall short [2]. Some examples where ANNs provide an effective solution include function approximation [3], time series signal prediction [4], correlation analysis [5], classification and pattern recognition [6], model predictive control [7] and deep reinforcement learning [8].

A fully connected, feed-forward ANN is a network of neurons organized in a number of *layers*. The neurons of the first layer are the *input* values. The neurons of the last layer are the *output* values. All other layers are called the *hidden* layers. During *activation,* the output (i.e., *y*) values are computed for some input (i.e., *x*) values. During activation, each neuron receives a *weighted* sum from neurons in the previous layer, makes a *bias* adjustment, applies an *activation function* and feeds its value forward. Hence, an ANN must be configured with a large set of weight/bias values. During *training*, a cost function is used to compute the error in the activation output of the ANN with respect to the target values. Through a *back-propagation* process, these error derivatives are used to adjust the weight/bias values at some rate (i.e., *learning rate*). Typically, a number (i.e., *batch*) of training passes are averaged prior to adjusting the main weight/bias values for better numerical stability. A comprehensive discussion of the activation and training techniques can be found here [9][10].

For general computing, a number of high-level modeling platforms exist that enable designers and scientists to rapidly capture ANNs and train/deploy them efficiently. These platforms are often available with bindings in high-level languages such as Python [11], Apache Spark [12], MATLAB/Simulink [13] and so on. Perhaps one of the best-known and highly optimized examples is the TensorFlow platform [14], which we will use as a benchmark in this work. While such platforms are appropriately well suited for desktop and cloud computing environments, they are highly onerous for use in embedded environments with limited compute and memory resources [15][16]. The Gravity compiler combines the abstraction and ease-of-use attributes of TensorFlow with a lightweight and resource optimized implementation that is well suited for resource constrained embedded devices.

The key contributions of this work include:

- Gravity, a compiler taking as input an abstract description of an ANN and generating an ANSI C compliant, dependency free and memory optimized executable ANN with training and inference functionality. Specifically, the generated code uses a succinct memory representation for weight/bias and intermediate calculations without reliance on a dynamic memory manager. The output is dependency free and does not require linking with external libraries.
- Gravity is constructed as a modern compiler consisting of an intuitive input language, an expressive Intermediate Representation (IR), a mapping to a Fictitious Instruction Set Machine (FISM) and a retargetable backend, making it an ideal research tool for exploring high-performance embedded software strategies in AI and Deep-Learning applications.
- Gravity is open source and freely available for community use and contributions. Moreover, all examples, data, validation and testing reported in this paper are released as part of the Gravity repository for complete reproducibility [17].

This paper describes the innerworkings of Gravity and demonstrates the benefits of using it to solve the MNIST handwriting digit recognition system [18][19].

ASPDAC '21, January 18–21, 2021, Tokyo, Japan
© 2021 Copyright is held by the owner/author(s).
ACM ISBN 978-1-4503-7999-1/21/01.
https://doi.org/10.1145/3394885.3431514

Directive	Req.	Description
.module *string-arg*	Y	Defines the name of the output C files to be the *string-arg*.
.prefix *string-arg*	N	Prepends the *string-arg* to every external C function generated by the Gravity compiler.
.optimizer *opt args*	N	Specifies the objective function optimization strategy. Default is stochastic gradient descent with a learning rate of 0.1[20].
.precision *type*	N	Specifies the numerical precision of the generated code. The type may be any of *float*, *double* or *fixed* [*whole-digits*, *frac-digits*]. The default is 32-bit IEEE floating point, i.e., *float*.
.costfnc *fnc*	N	Specifies the loss function between the ANN prediction and training labels. The *fnc* may be any of *quadratic*, *exponential* or *cross_entropy*. The default is the cross entropy [21].
.batch *int-expr*	N	Specifies the training batch size. Default is 1.
.input *int-expr*	Y	Specifies the ANN input layer dimension. The *int-expr* is any valid positive integer expression.
.output *int-expr act-fnc*	Y	Specifies the ANN output layer dimension and activation function. The *int-expr* is any valid positive integer expression. The *act-fnc* is one of *ReLU*, *linear*, *SoftMax* or *Sigmoid*.
.hidden *int-expr act-fnc*	Y	Specifies an ANN hidden layer dimension and activation function. The *int-expr* is any valid positive integer expression. The *act-fnc* is one of *relu*, *linear*, *softmax* or *sigmoid*. At least one hidden layer is mandatory. The ordering of the hidden layers will follow the order in which they are defined in the source file.

Table 1: The Gravity Language Specification.

II. The Gravity Compiler

In this section, we describe each of the main components of the Gravity compiler. These include the input language and lexical analyzer/parser that generate an Intermediate Representation (IR), the Artificial Neural Network (ANN) compiler that generates the output program for a Fictitious Instruction Set Machine (FISM) and the final stage of generating the ANSI C code using a retargetable backend. Then, we describe the memory architecture used by Gravity to accommodate the weight/bias as well as intermediate values.

A. Language Analyzer/Parser & Intermediate Representation

Gravity takes as input a simple text file description of the desired ANN. By convention, a program written for Gravity will have a `*.g` extension and contain a number of directives. Other than directives, the only other permissible content may be white spaces or comments, which begin with the "//" and extend to the end of the line. Table 1 lists all the available directives. Listing 1 gives a complete Gravity program that describes the ANN model used in our MNIST handwriting digit recognition system.

```
.module "mnist"; // output: mnist.[h|c]
.prefix "mnist";
.optimizer sgd 0.1;
.precision float;
.costfnc cross_entropy;
.batch 8;
.input 28 * 28;
.output 10 softmax;
.hidden 100 relu; // first hidden layer
.hidden 100 relu; // second hidden layer
```

Listing 1. The Gravity description of the MNIST ANN.

Gravity uses lex/yacc to generate the lexical analyzer and parser front-end. This front-end tokenizes the input, checks for grammatical correctness and calls a set of backend functions for each language production to construct the IR object. The IR object is a C `struct` type containing nested elements that closely correspond to the directives listed in Table 1. In particular, the top-level object contains a member array object called "nodes" that recursively defines the input, hidden layers 1, 2, … and output layer specification. Finally, additional sanity checks and ANN model correctness steps are carried out over the IR object. The fully validated IR object is then passed on to the ANN compiler to generate a number of programs for a virtual machine with a fictitious set of instructions.

B. Artifical Neural Network Compiler

Gravity translates the ANN model (i.e., the IR) to three distinct programs, each defined as a sequence of high-level instructions for FISM, namely: *activate*, *propagate* and *train*. Think of these programs as procedures in high-level languages.

The *activate* program implements an inference pass on a fully connected ANN having layers 1 (input), 2 … *N–1* (hidden) and *N* (output). The *activate* program computes $a^{1 \ldots N}$ as shown:

$$a^1 = x$$

$$a^l = \sigma\left(\sum_{i=1}^{K^l} \sum_{j=1}^{K^{l-1}} a_j^{l-1} \times W_{i,j}^l + B_i^l\right)$$

Here, x is the ANN input vector and W/B represent the weights and biases. Superscripts reference the corresponding layer and subscripts reference the corresponding neurons within a layer. The indices of $W_{i,j}$ represent the weights on the edge between the source i and destination j neurons. Finally, K^l denotes the number of neurons at the given layer l. As a last step, an activation function (i.e., σ) is applied to the weighted average. The *activate* program is used for inference as well as for training. For inference, the ANN model output is a^N. Hence the generated C pseudocode for forward inference is defined as:

```
inference( $x \rightarrow y$ )
  activate( $x \rightarrow a^{1 \ldots N}$ )
  $y = a^N$
```

The *propagate* program computes a single back-propagation pass using the previously computed $a^{1 \ldots N}$ activations. Specifically, given y as a sample output label, the algorithm starts from the last layer N, computes a loss function and uses the chain-rule to compute partial derivatives for each of the layers, storing the results in $d^{N \ldots 2}$. A full description of the back-propagation algorithm can be found here [9][10]. The ANN compiler generates a sequence of high-level instructions for FISM that precisely compute a set of derivatives $d^{N \ldots 2}$. The derivatives $d^{N \ldots 2}$ are used to compute adjustments, a batch at a time, to the W and B parameters using the specified optimization strategy (e.g., stochastic gradient descent). Hence the generated C pseudocode for the *train* program is as follows:

```
train( $x_{1 \ldots S}, y_{1 \ldots S} \rightarrow W,B$ )
   for $s = 1 \ldots S$ /* batch_size */
      activate( $x_s \rightarrow a$ )
      propagate( $a, y_s \rightarrow +W', +B'$ )
   $W/B = W/B$ - learning_rate * $W'/B' \div S$
```

In the above pseudocode, weight/bias adjustments are averaged as *W'/B'* and then applied to the actual *W*/*B* parameters using the specified learning rate.

The ANN compiler produces intermediate code using the FISM instructions for each of the three programs: *activate*, *propagate* and *train*. These instructions are summarized in Table 2. Listing 2 shows the generated FISM instructions for the *activate* program of the ANN model used in our MNIST handwriting digit recognition system.

Instruction	Description
RET	Program ends and returns void.
RETARG *a*	Program ends and returns *a*.
BATCHLOOP *N*	Executing a training loop with *N* iterations.
RANDOM *a*, *N*, *R*	$a_i = random(-R\ to + R) \mid i = 1 \ldots N$
CLEAR *a*, *N*	$a_i = 0 \mid i = 1 \ldots N$
COPYX *a*, *N*	$a_i = x_i \mid i = 1 \ldots N, x\ is\ the\ ANN\ input$
MAC1 *a*, *b*, c, *N*, *M*	$a_i = \sum_{i=1}^{N}\sum_{j=1}^{M} b_{i,j} \times c_j$
MAC2 *a*, *b*, c, *N*, *M*	$a_i = \sum_{i=1}^{M}\sum_{j=1}^{N} b_{j,i} \times c_j$
MAC3 *a*, *b*, c, *N*, *M*	$t = a,\ \ a_{i,j} = \sum_{i=1}^{N}\sum_{j=1}^{M} t_{i,j} + b_i \times c_j$
MAC4 *a*, *b*, *C*, *N*	$a_i = a_i + b_i \times C \mid i = 1 \ldots N$
ADD *a*, *b*, *N*	$a_i = a_i + b_i \mid i = 1 \ldots N$
SUBY *a*, *N*	$a_i = a_i - y_i \mid i = 1 \ldots N, y\ is\ the\ ANN\ output$
RELU *a*, *N*	$a_i = \begin{cases} 0, & a_i < 0 \\ a_i, & a_i \geq 0 \end{cases} \mid i = 1 \ldots N$
LINEAR *a*, *N*	$a_i = a_i \mid i = 1 \ldots N$
SOFTMAX *a*, *N*	$a_i = softmax(a, a_i) \mid i = 1 \ldots N$
SIGMOID *a*, *N*	$a_i = sigmoid(a, a_i) \mid i = 1 \ldots N$
RELUD *a*, *b*, *N*	$a_i = \begin{cases} 0, & b_i < 0 \\ a_i, & b_i \geq 0 \end{cases} \mid i = 1 \ldots N$
SOFTMAXD *a*, *b*, *N*	$a_i = softmax'(a, a_i - b_i) \mid i = 1 \ldots N$
SIGMOIDD *a*, *b*, *N*	$a_i = sigmoid'(a, a_i - b_i) \mid i = 1 \ldots N$

Table 2: The FISM instructions.

```
COPYX <float>   @716880 784
MAC1 <float>    @720016 @0 @716880 100 784
ADD <float>     @720016 @313600 100
RELU <float>    @720016 100
MAC1 <float>    @720816 @314000 @720016 100 100
ADD <float>     @720816 @354000 100
RELU <float>    @720816 100
MAC1 <float>    @721616 @354400 @720816 10 100
ADD <float>     @721616 @358400 10
SOFTMAX <float> @721616 10
RETARG <float>  @721616
```

Listing 2. ANN Compiler FISM output (*activate* program).

Listing 2 shows a sequence of instructions where each instruction is annotated with the desired precision, in this case 32-bit IEEE floating point. Depending on the user specified precision directive, the ANN compiler will generate appropriate instructions, including 64-bit IEEE floating point or fixed-point integer arithmetic with a user specified precision for the whole and fractional parts of the real value.

Furthermore, Listing 2 shows the arguments to the FISM instructions which are either a numerical address of the memory containing the operand vectors (i.e., those starting with the @ symbol) or integers that define the vector dimensions as shown in Table 2. A FISM program describes a dataflow in terms of a sequential set of instructions. Hence, jumps and branches are not allowed. However, each one of the FISM instructions, as described in Table 2, is in turn implemented with some looping internal structure. These looping structures have two attributes that make them ideal for further optimization: (1) the number of loop iterations are always a priori known and (2) the loop nests access memory sequentially.

Finally, the **BATCHLOOP** *N* instruction deserves a special mention. This instruction is used to emulate the batch training loop of the *train* program (see above). Semantically, it calls the *activate* followed by the *propagate* programs *N* times.

C. Retargtable Backend

For each of the FISM programs (i.e., *activate*, *propagate* and *train*), Gravity's retargetable backend is responsible for generating the output ANSI C code. Currently, the ANSI C generator is the only available target, however, other backends (e.g., OpenCL and OpenGL for use with FPGA/GPU acceleration [22][23]) are being developed.

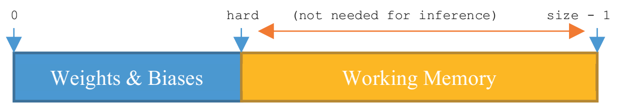

Figure 1: Memory Utilization of Gravity Generated ANSI C Code

The ANSI C backend applies a template for each of the FISM instructions listed in Table 2 to generate a corresponding output. For example, the template for the **MAC1** instruction is shows in Listing 3.

```
{ /* MAC1 */
  %type1 *z = (%type1 *)( m_ + %offset1 );
  const %type1 *A = (const %type1 *)( m_ + %offset2 );
  const %type1 *B = (const %type1 *)( m_ + %offset3 );
  %type2 i, j;
  for (i=0; i<%bound1; ++i) {
    z[i] = 0.0;
    for (j=0; j<%bound2; ++j) {
      z[i] += A[i * %bound2 + j] * B[j];
    }
  }
}
```

Listing 3. ANSI C backend template for MAC1 instruction.

Here, *type1* is derived from the ANN precision specification and *type2* is automatically optimized to be sufficiently large to accommodate the loop iterations. The *offset1*, *offset2* and *offset3* give the memory address of the operand vectors and are extracted from the corresponding FISM instruction. Likewise, *bound1* and *bound2* are extracted from the corresponding FISM instructions. The generated ANSI C code for the MNIST handwriting digit recognition system is shown in Listing 5 (*activate* program).

These ANSI C templates are carefully designed to provide an optimizing C compiler all the hints necessary to generate highly optimized machine code and are void of safety hazards.

D. The Gravity Memory Model

Gravity packs the various vectors needed to train or activate the ANN in a contiguous memory region. Moreover, this contiguous memory region is composed of two sub regions, starting with the memory address [0 … *hard* – *1*] and [*hard* … *size* – *1*]. For a particular ANN specification, the *hard* and *size* values are computed by Gravity and can be queried by the calling application to supply the required memory as a single byte array. The memory utilization of the generated code will depend on the precision specification as well as the number of ANN layers and neurons. However, the memory utilization is both minimal and deterministic.

The region of contiguous memory from [0 … *hard* – 1] contains the weight/bias values plus a small amount of working memory needed for inference. No other memory is allocated or used by the generated ANSI C code. Moreover, this first region of memory is serializable. In other words, a calling application can store and retrieve this region of memory as needed. This is how a trained ANN can be persisted for subsequent inference. The region of contiguous memory from [*hard* … *size* – 1] is used during training to hold intermediate calculations.

The memory addresses referenced in Listing 2 (i.e., those starting with the @ symbol) denote an offset within this contiguous memory region.

Figure 1 pictorially depicts the memory layout of Gravity generated ANSI C code.

Another capability of the Gravity compiler is that it can generate code with any of the **float**/**double** precision IEEE or user specified fixed point arithmetic with the desired number of bits dedicated to the whole part as well as the fractional part (see ***.precision*** directive in Table 1).

E. Invoking the Gravity Compiler

The Gravity compiler is open source software distributed under the GNU General Public License v3.0. The compiler itself is written in ANSI C and available as a public GIT repo [17]. It has been compiled on macOS as well as different Unix/Linux flavors. Other than lex/yacc, there are no external library dependencies when building the Gravity source code. Gravity can be used in two modes: (1) as a traditional compiler and (2) as a just-in-time (JIT) compiler.

The traditional compiler mode includes modeling the ANN in the Gravity language and using the gravity compiler to generate the `*.[c|h]` files. These files are then compiled and linked with the target application. The generated MNIST digit recognition function signatures are shown in Listing 4.

```
int mnist_version(void);
size_t mnist_memory_size(void);
size_t mnist_memory_hard(void);
void mnist_initialize(void *m);
void *mnist_activate(void *m, const void *x);
void mnist_train(void *m, const void *x, const void *y);
```

Listing 4. Generated ANSI C code function signatures.

Here, `version()` returns the version of the Gravity compiler used, `memory_size()` returns the size of the contiguous memory needed for inference and training, `memory_hard()` returns the size of the contiguous memory needed for inference only, `initialize()` assigns random weight/bias values to a contiguous memory, `activate()` runs inference on input `x` using the weight/bias values stored in the contiguous memory `m` and `train()` trains a batch of input/output pairs (i.e., `x` and `y`) and updates the weights in `m`.

When building Gravity, in addition to the compiler executable, `libgravity.[a|so]` are generated. An application can link with these libraries and use Gravity facilities in JIT mode. Specifically, Gravity can be embedded within an application and an executable ANN can be created dynamically using the function show in Listing 6.

```
g_t g_open(const char *optimizer,
           const char *precision,
           const char *costfnc,
           const char *batch,
           const char *input,
           const char *output,
              /* hidden */ ...);
```

Listing 6. Gravity compiler used in JIT mode.

Here, `g_open()` is used to invoke the Gravity compiler to generate an ephemeral ANSI C file, which in turn is compiled on-the-fly into machine code and dynamically linked (using `dlopen`) with the calling process. This function's arguments closely resemble the directives stated in Table 1. A returned handle is thereafter used with functions identical to those listed in Listing 4. The JIT mode is ideal for desktop or cloud use cases where a highly optimized ANN executable model is desired. One such use case is in design space exploration where a large number of hyper parameters are explored in parallel.

Experiments and Validation

We set forth to establish the efficacy of Gravity in a series of experiments and validation benchmarks. To do so, we utilized Gravity as the core component in solving the MNIST handwriting digit recognition system [18][19].

The MNIST dataset is composed of 60,000 28×28 8-bit/grayscale images of handwritten digits (i.e., 0 through 9) for training and 10,000 additional images, having the same attributes, for testing. Furthermore, the datasets include, for each image, a label that identifies the depicted digit in the corresponding image (i.e., a number between 0 and 9). The

```
static float *_activate_(char *m_, const float *x_) {
  { /* COPYX */
    memcpy(m_ + 716880, x_, 784 * sizeof (float));
  }
  { /* MAC1 */
    float *z = (float *)( m_ + 720016 );
    const float *A = (const float *)( m_ + 0 );
    const float *B = (const float *)( m_ + 716880 );
    uint32_t i, j;
    for (i=0; i<100; ++i) {
      z[i] = 0.0;
      for (j=0; j<784; ++j) {
        z[i] += A[i * 784 + j] * B[j];
      }
    }
  }
  { /* ADD */
    float *za = (float *)( m_ + 720016 );
    const float *B = (const float *)( m_ + 313600 );
    uint32_t i;
    for (i=0; i<100; ++i) {
      za[i] += B[i];
    }
  }
  { /* RELU */
    float *za = (float *)( m_ + 720016 );
    uint32_t i;
    for (i=0; i<100; ++i) {
      if (0.0 >= za[i]) {
        za[i] = 0.0;
      }
    }
  }
  { /* MAC1 */
    float *z = (float *)( m_ + 720816 );
    const float *A = (const float *)( m_ + 314000 );
    const float *B = (const float *)( m_ + 720016 );
    uint32_t i, j;
    for (i=0; i<100; ++i) {
      z[i] = 0.0;
      for (j=0; j<100; ++j) {
        z[i] += A[i * 100 + j] * B[j];
      }
    }
  }
  { /* ADD */
    float *za = (float *)( m_ + 720816 );
    const float *B = (const float *)( m_ + 354000 );
    uint32_t i;
    for (i=0; i<100; ++i) {
      za[i] += B[i];
    }
  }
  { /* RELU */
    float *za = (float *)( m_ + 720816 );
    uint32_t i;
    for (i=0; i<100; ++i) {
      if (0.0 >= za[i]) {
        za[i] = 0.0;
      }
    }
  }
  { /* MAC1 */
    float *z = (float *)( m_ + 721616 );
    const float *A = (const float *)( m_ + 354400 );
    const float *B = (const float *)( m_ + 720816 );
    uint32_t i, j;
    for (i=0; i<10; ++i) {
      z[i] = 0.0;
      for (j=0; j<100; ++j) {
        z[i] += A[i * 100 + j] * B[j];
      }
    }
  }
  { /* ADD */
    float *za = (float *)( m_ + 721616 );
    const float *B = (const float *)( m_ + 358400 );
    uint32_t i;
    for (i=0; i<10; ++i) {
      za[i] += B[i];
    }
  }
  { /* SOFTMAX */
    float *za = (float *)( m_ + 721616 );
    float max=za[0], sum=0.0;
    uint32_t i;
    for (i=1; i<10; ++i) {
      if (max < za[i]) {
        max = za[i];
      }
    }
    for (i=0; i<10; ++i) {
      za[i] -= max;
      sum += (float)exp(za[i]);
    }
    for (i=0; i<10; ++i) {
      za[i] = (float)exp(za[i]) / sum;
    }
  }
  { /* RETARG */
    return (float *)( m_ + 721616 );
  }
}
```

Listing 5: Generated ANSI C code (*activate* program).

	Accuracy	Train-Time (usec / sample)	Test-Time (usec / sample)	Train/Test-Memory (MB)
GravityC	0.967	124	18.3	0.722 / 0.358
TensorFlow	0.971	154	39.7	209 / ---
TVM	0.966	211	67.9	34.5 / ---

Table 5: Gravity performance vs TensorFlow/TVM on iMac.

	Accuracy	Train-Time (usec / sample)	Test-Time (usec / sample)	Train/Test-Memory (MB)
GravityC	0.967	2060	139	0.722 / 0.358
TensorFlow	FAIL	FAIL	FAIL	FAIL
TVM	0.965	3019	328	33.5 / ---

Table 3: Gravity performance vs TensorFlow/TVM on Raspberry Pi 3.

	Accuracy	Train-Time (usec / sample)	Test-Time (usec / sample)	Train/Test-Memory (MB)
GravityC	0.967	475	33.3	0.722 / 0.358
TensorFlow	0.971	644	111	209 / ---
TVM	0.965	911	78.4	38.2 / ---

Table 4: Gravity performance vs TensorFlow (Raspberry Pi 4).

MNIST problem is to build a classifier capable of being trained with the 60,000 image/label pairs (i.e., training dataset) and tested with the additional 10,000 image/label pairs (i.e., the test dataset) for accuracy.

As a reference implementation, we chose the Python TensorFlow platform (data and source code available at the Gravity repository [17]). We modeled the MNIST ANN as having 784 inputs, two hidden layers (i.e., 100 neurons per layer and ReLU activation) and 10 output neurons using SoftMax activation. Furthermore, we used a stochastic gradient descent optimization strategy with a learning rate of 0.1. Our cost function for training was set to cross-entropy. Tuning of this ANN would require setting 89,400 weight and 220 bias values.

For performance measurements, we ran the TensorFlow as well as TVM [27] implementations for 4 training/testing epochs on a desktop computer. A training epoch consisted of feeding batches of 8 images as input and using the known labels as output for the entirety of the 60,000 image/label pairs. Testing consisted of feeding the 10,000 test images as input and checking the output for correctness against the known labels. Likewise, we used Gravity to generate an identical ANN and combined it with a small C driver application to execute precisely as our TensorFlow/TVM implementations. Our Gravity description of the ANN is shown in Listing 1. We preloaded the training/test datasets into RAM in both TensorFlow/TVM and Gravity/C versions to exclude the effects of dataset load time from our measurements.

In our benchmarking, we tracked the following metrics: *accuracy* (i.e., measured as the number of correct test activations divided by the total number of test images, namely, 10,000), *training-time* (i.e., the time it takes per a single training backpropagation pass), *activation-time* (i.e., the time it takes per a single inference pass) and *train/test-memory* (i.e., the amount of RAM necessary to train and activate, respectively).

A. *Gravity Generated C Code vs. TensorFlow/TVM Models*

For the first experiment, our machine consisted of a 4 GHz Quad-Core Intel Core i7 with 32 GB 1867 MHz DDR3 running CentOS Linux release 8.0.1905. The software environment included GCC 8.2.1, Python 3.6.8 and TensorFlow 2.0. Results are given in Table 3. Table 3 shows that despite TensorFlow's highly optimized engine, the Gravity generated ANSI C code performed 20% better during training and 2.4x better during inference. More importantly, the memory utilization of the Gravity generated code was approximately 300x lower. Similarly, Gravity outperformed TVM by a considerable margin.

We repeated the tests on a Raspberry Pi 3 Model B+ having a 1.4 GHz 64-bit Quad-Core ARM Cortex-A53 processor running Raspbian 9.11. This system had 1 GB of RAM. The software environment included GCC 6.3.0, Python 3.5.3 and TensorFlow 1.14.0 as well as TVM. Results are given in Table 4. Table 4 shows that Gravity was successful at running the MNIST benchmark on limited resources with predictable memory utilization and reasonably performant training and inference latencies. On the other hand, TensorFlow failed to execute this benchmark due to out-of-memory faults, despite numerous attempts to tune and optimize the TensorFlow model. TVM was able to execute on this device, but it only achieved 50% training rate compared to Gravity and 2.4x reduction in inference times.

To address the above mentioned TensorFlow failure, we opted for a more powerful Raspberry Pi 4 Model B 2019 having a 1.5 GHz 64-bit Quad-Core ARM Cortex-A72 processor running Raspbian Gnu/Linux 10 (buster). This system had 4 GB of RAM. The software environment included GCC 8.3.0, Python 3.7.3 and TensorFlow 1.14.0. Results are given in Table 5. This time, we were able to execute the MNIST benchmark using TensorFlow. Nevertheless, Gravity generated ANSI C code outperformed TensorFlow in every aspect averaging 35% better training times, 4.2x better testing times, and 289x lower memory utilization.

For all TensorFlow tests, we note that we measured the `libtensorflow-framework.so.2` resident (i.e., physical) memory only to better isolate the ANN memory utilization (the total Python process memory utilization exceeded 1 GB). There are no facilities in TensorFlow to limit the memory utilization for inference only. Furthermore, the accuracy of both TensorFlow and Gravity tracked closely and the small variance in accuracy was discovered to be a function of different random weight/bias assigned values during initialization.

Averaging all Gravity vs. TensorFlow data, we measured a 33% speedup in training, 2.5x speedup in inference, and 300x reduction in memory utilization.

B. *Gravity Generated C Code for a Deeper ANN*

We repeated our previous experiments on the MNIST dataset but this time changed the ANN architecture from 784:100:100:10 to 784:50:50:50:50:10. Specifically, we changed the ANN from 4 layers (inclusive of input/output) to 6 layers with each hidden layer having half as many neurons. We re-evaluated the same set of performance metrics (i.e., accuracy, train-time and test-time) for Gravity generated code vs. TensorFlow and TVM. Our results are summarized in Table 6.

As expected, the running time improved, as the overall capacity computational load of the deeper ANN was reduced. Of note is that Gravity generated C code showed better execution time improvement in both training and inference with respect to TensorFlow and TVM. Specifically, Gravity training time was reduced by 37% while TensorFlow and TVM was reduced by 25%. Similarly, Gravity inference time was reduced by 56% while TensorFlow and TVM was reduced by 47/45% (TensorFlow/TVM).

	Accuracy	Train-Time (usec / sample)	Test-Time (usec / sample)	Train/Test-Memory (MB)
GravityC	0.955	79.1	8.00	0.384 / 0.190
TensorFlow	0.954	115	21.2	171 / ---
TVM	0.931	158	37.5	19.6 / ---

Table 6: Gravity performance of Deep ANN vs TensorFlow/TVM on iMac.

C. Gravity Generated C Code on a Microcontroller

A main objective in the design of Gravity was to create an automated and rapid software synthesis methodology for embedded applications. To test this, we designed an embedded device with a single AVR ATMEGA1284-PU microcontroller running at 20 MHz. This microcontroller is a tiny 8-bit device without support for floating point arithmetic [24]. Our microcontroller had 128 KB of program memory and 16 KB of RAM. We attached a 16 GB SD card to our microcontroller using its SPI interface to hold our training and test datasets (i.e., approximately 50 MB of data). We added 1 MB of external SRAM to serve as the working memory (8 × AN1245 serial SRAM). We ported our Gravity implementation of MNIST to this embedded device (i.e., modified the C driver application to load the datasets from the SD card) and successfully ran a number of training and test epochs to measure performance. We used GCC as a cross-compiler to build the AVR binary.

For this embedded device, the total MNIST code size, including the driver application and the 32-bit floating point emulation library was approximately 107 KB. Our embedded device was able to execute a single training pass in approximately 1.63 sec/sample. The test pass took approximately 0.486 sec/sample. While slow compared to our powerful desktop, these numbers are remarkable given the limited capabilities of our tiny microcontroller!

We intentionally chose a tiny 8-bit microcontroller in this experiment to demonstrate the enabling capabilities of Gravity. It would be extremely difficult, if not impossible, to port TensorFlow to an 8-bit AVR! While a competent engineer can program the same ANN in C by hand, gravity automates the process, generates code in a fraction of a second, does not require domain expertise and outputs a correct by construction executable model from an abstract ANN description. This is the value added to the design automation community.

Related Work

We have already introduced a number of platforms, including TensorFlow, which provide a rich set of API for capturing ANNs at a high level of abstraction [11][12][13][14]. While highly efficient and easy to use, these platforms are very demanding in terms of resource utilization and unavailable on embedded devices with limited capabilities. A well-known attempt at addressing ANNs for embedded devices is the FANN [25] library. While lightweight, FANN is not an ANN compiler and is mostly optimized for inference rather than training. FANN has a higher memory and execution overhead and requires porting to a target that is more onerous in nature. Similarly, TensorFlow Lite is a designed for deployment on IoT and mobile devices. TensorFlow Lite is an open source deep learning framework for machine learning models. Unlike Gravity, TensorFlow Lite is intended for on-device inference only. Gravity, on the other hand can be used for efficient training as well. As with other library-based solution, TensorFlow Lite is a one size fits all attempt and lacks the ultra-low resource utilization design principles of Gravity.

Perhaps one of the most significant contribution in this area is TVM, a compiler that exposes graph-level and operator-level optimizations to provide performance portability to deep learning workloads across diverse hardware back-ends. TVM solves optimization challenges specific to deep learning, such as high-level operator fusion, mapping to arbitrary hardware primitives, and memory latency hiding. It also automates optimization of low-level programs to hardware characteristics by employing a novel, learning-based cost modeling method for rapid exploration of code optimizations [27]. While TVM has a similar purpose as Gravity, its footprint, in terms of code size, is considerably larger than Gravity and has lower performance than Gravity, as illustrated in our experiments.

Another relevant art, called Glow is a machine learning compiler for heterogeneous hardware [28]. Specifically, Glow, is a pragmatic approach to compilation that enables the generation of highly optimized code for multiple targets. Glow lowers the traditional neural network dataflow graph into a two-phase strongly typed intermediate representation. The high-level intermediate representation allows the optimizer to perform domain-specific optimizations. As with many other approaches, the Glow design flow is targeted toward research rich systems and is unable to generate efficient code on embedded, low-memory systems.

Another class of work in this area is the building of custom compilers for neuromorphic hardware [29][30][31]. These compilers assume the existence of some highly specialized processing element and generate code that is intended to run on these hardware accelerators. In contrast, Gravity generates ANSI C code for standard processors, including those without floating point support. Moreover, these compilers require a pre-trained ANN to function correctly [29].

Hardware implementation of ANNs have received tremendous attention, particularly for low latency applications. A good summary of such techniques can be found here [32]. Gravity can be retargeted to generate high-level circuit description with OpenCL as a backend.

Conclusion

This paper introduced Gravity, an Artificial Neural Network to ANSI C compiler that is particularly good at generating software for embedded systems. Gravity automates the generation of tight and lightweight code for resource constrained embedded processors. This open-source project is intended as a useful tool for engineers who are designing AI/ML-based embedded devices as well as researchers who are exploring high-performance embedded software strategies in AI and Deep-Learning applications. We demonstrated the usefulness of the Gravity compiler with the MNIST handwriting digit recognition system in an embedded environment that is capable to train and activate a relatively large ANN on limited memory and compute resources. We measured a 300x reduction in memory usage, 2.4x speedup in inference and 20% speedup in training compared to TensorFlow. We are currently working on adding OpenCL and OpenGL backends to Gravity. We like to explore more sophisticated data precision management systems. Future work can include using Gravity as the kernel of a design space exploration algorithm to optimize the ANN hyper-parameter space.

Acknowledgments

This work was supported in part by the National Science Foundation under NSF grant number 1563652.

References

[1] K. Uma Rao, "Artificial intelligence and neural networks," Pearson Education, 2011, ISBN 13: 9788131759653.

[2] H. White, "Artificial neural networks: approximation and learning theory," Blackwell Publishers, 1992, ISBN:1557863296.

[3] Z. Zainuddin and O. Pauline, "Function approximation using artificial neural networks," WSEAS Transactions on Mathematics, 7(6), pp. 333-338, 2008.

[4] G.P. Zhang, B.E. Patuwo and M.Y. Hu, "A simulation study of artificial neural networks for nonlinear time-series forecasting," Computers & Operations Research, 28(4), pp. 381-396, 2001.

[5] P.L. Lai and C. Fyfe, "Canonical correlation analysis using artificial neural networks," In proceedings of the ESANN, pp. 363-368, 1998.

[6] J.K. Basu, D. Bhattacharyya and T.H. Kim, "Use of artificial neural network in pattern recognition," International Journal of Software Engineering and its Applications, 4(2), 2010.

[7] M. Amir, T. Givargis and F. Vahid, “Switching predictive control using reconfigurable state-based model,” ACM Transactions on Design Automation of Electronic Systems (TODAES), 24(1), pp. 1-21, 2018.

[8] H. Buini, S. Peter and T. Givargis, “Adaptive embedded control of Cyber-physical systems using reinforcement learning,” IET Cyber-Physical Systems: Theory & Applications (IET), 2(3), pp. 127-135, 2017.

[9] B.J. Wythoff, “Backpropagation neural networks,” Chemometrics and Intelligent Laboratory Systems, 18(2), pp. 115-155, 1993.

[10] R. Hecht-Nielsen, “Theory of the backpropagation neural network,” In proceedings of the Neural Networks for Perception, pp. 65-93, 1992.

[11] The Python software fondation, https://www.python.org.

[12] The Apache software foundation, https://spark.apache.org.

[13] MathWorks, https://www.mathworks.com/products/matlab.html.

[14] TensorFlow, https://www.tensorflow.org.

[15] N.D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi and F. Kawsar, “Accelerated deep learning inference for embedded and wearable devices using DeepX,” In proceedings of the 14th ACM Annual International Conference on Mobile Systems, Applications, and Services Companion, pp. 109-109, 2016.

[16] W. Meng, Z. Gu, M. Zhang and Z. Wu, “Two-bit networks for deep learning on resource-constrained embedded devices,” arXiv preprint arXiv:1701.00485, 2017.

[17] The Gravity compiler, www.github.com/givargis/gravity.

[18] Y. LeCun, C. Cortes and C. Burgest, “The MNIST database of handwritten digits,” http://yann.lecun.com/exdb/mnist.

[19] L. Deng, “The MNIST database of handwritten digit images for machine learning research [best of the web],” IEEE Signal Processing Magazine, 29(6), pp. 141-142, 2012.

[20] L. Bottou, “Stochastic gradient learning in neural networks,” In proceedings of the Neuro-Nımes, 91(8), pp. 12-18, 1991.

[21] A.R. Barron, “Statistical properties of artificial neural networks,” In proceedings of the 28th IEEE Conference on Decision and Control, pp. 280-285, 1989.

[22] The Open Computing Language, www.khronos.org/opencl.

[23] The Open Graphics Library, www.opengl.org.

[24] Microchip Technology Inc, www.microchip.com.

[25] S. Nissen, “Implementation of a fast artificial neural network library (fann),” Report, Department of Computer Science University of Copenhagen (DIKU), 2003.

[26] TensorFlow Lite, https://www.tensorflow.org/lite.

[27] Chen T, Moreau T, Jiang Z, Zheng L, Yan E, Shen H, Cowan M, Wang L, Hu Y, Ceze L, Guestrin C. TVM: An automated end-to-end optimizing compiler for deep learning. USENIX Symposium on Operating Systems Design and Implementation OSDI, pp. 578-594, 2018.

[28] Rotem N, Fix J, Abdulrasool S, Catron G, Deng S, Dzhabarov R, Gibson N, Hegeman J, Lele M, Levenstein R, Montgomery J. Glow: Graph lowering compiler techniques for neural networks. arXiv preprint arXiv:1805.00907, 2018.

[29] Y. Ji, Y. Zhang, W. Chen and Y. Xie, “Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler,” In proceedings of the ACM SIGPLAN Notices, 53(2), pp. 448-460, 2018.

[30] A. Ankit, I.E. Hajj, S.R. Chalamalasetti, G. Ndu, M. Foltin, R.S. Williams, P. Faraboschi, W.M.W Hwu, J.P. Strachan, K. Roy and D.S. Milojicic, “PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference,” In proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 715-731, 2019.

[31] H. Fang, A. Shrestha, Z. Zhao and Y. Wang, “A general framework to map neural networks onto neuromorphic processor,” In proceedings of the 20th International Symposium on Quality Electronic Design (ISQED), pp. 20-25, 2019.

[32] A. Ardakani, F. Leduc-Primeau, N. Onizawa, T. Hanyu and W.J. Gross, “VLSI implementation of deep neural network using integral stochastic computing,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 25(10), pp.2688-2699, 2017.