A LANGUAGE AND COMPILER VIEW ON DIFFERENTIABLE PROGRAMMING

Fei Wang, **Xilun Wu**, Tiark Rompf
Purdue University, Department of Computer Science

Background

Current and emerging Deep Learning architectures call for an expressive high-level programming style with end-to-end differentiation and for a high-performance implementation at the same time. But the current generation of deep learning frameworks either limits expressiveness and ease of use for increased performance (e.g., TensorFlow) or vice versa (e.g., PyTorch). We demonstrate that a “best of both worlds” approach is possible, based on multi-stage programming and delimited continuations, two orthogonal ideas firmly rooted in programming languages research.

Objectives

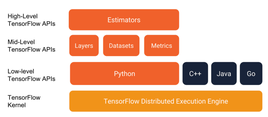
Our goal is to build a lightweight Deep Learning framework in form of a Scala library. We propose that: (1) Since the computation graphs play the similar role as AST in MSP, the explicit stage in existing frameworks can be eliminated. (2) Delimited Continuations can be used to implement Reverse-mode Automatic Differentiation in a concise (and functional) fashion. (3) MSP and Delimited Continuations can be combined in one framework.

Existing ML frameworks

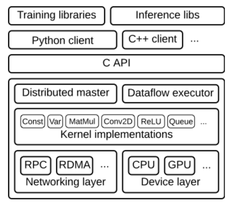
Popular deep learning frameworks rely on computation graphs to enable efficient automatic computation of gradients through reverse-mode automatic differentiation (AD). The computation graphs can be defined statically or built dynamically. TensorFlow and PyTorch are two representatives of these two styles.

TensorFlow is a cross-platform (backend) library which offers C API for clients in different languages:

- Define computation graph explicitly
- Multi-level hierarchy which hides low-level details well



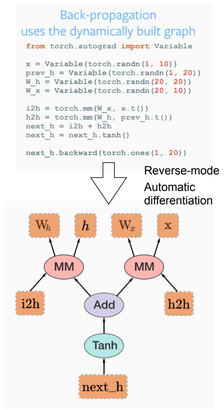
(a1) Multi-level API support of TensorFlow



(a2) Architecture of TensorFlow

PyTorch is a more lightweight and flexible Python library:

- Build graph dynamically using Tape based Autograd.
- Better integration of each part.



(a3) Tape based Autograd of PyTorch

Key idea

Gradient computation lies at the core of Neural Network. Automatic differentiation is a concise and efficient way to compute gradients without passing values among operators. We present a concise Reverse-mode Automatic-differentiation in functional style with Delimited Continuations.

```
class NumF(val x: Double, val d: Double) {
  def *(that: NumF) =
    new NumF(x * that.x,
      this.d * that.x + that.d * this.x)
  ...
}

a + a * a
```

```
class NumR(val x: Double, var d: Double) {
  def *(that: NumR) = { (k: NumR=>Unit) =>
    val y = new NumR(x * that.x, 0.0); k(y)
    this.d += that.x * y.d
    that.d += this.x * y.d
  }
  ...
}
(a * a)(aa => (aa + a)(aaa => aaa.d = 1.0))
```

```
class Num(val x: Double, var d: Double) {
  def *(that:Num) = shift {(k:Num=>Unit)=>
    val y = new Num(x * that.x, 0.0); k(y)
    this.d += that.x * y.d
    that.d += this.x * y.d
  }
  ...
}
reset { (a + a * a).d = 1.0 }
```

Figure 1: Automatic Differentiation in Scala: (a) forward mode, (b) reverse mode in continuation- passing style (CPS), (c) reverse mode using delimited continuations, with shift/reset operators.

Our prototype also uses the Lightweight Modular Staging (LMS) framework (Rompf & Odersky, 2010) to optimize the user-defined model and generate C++ code which is then compiled to binary and execute. Unlike PyTorch (relies on a tape structure) nor TensorFlow (relies on a computation graph), our prototype defines model without any explicit auxiliary data structures.

```
class Num(val x: Rep[Double], var d: Rep[Double])
{...}
def squash(x: Num) = {
  // Divide by 2.0 until less than 1.0
  var temp = x
  while (temp > 1.0) temp = 0.5 * temp
  temp
}
```

```
void squash_loop(double temp_x, double &temp_d) {
  if (temp_x > 1.0) {
    double y_d = 0.0;
    squash_loop(0.5 * temp_x, y_d); // recursive call, update y_d
    temp_d += 0.5 * y_d;
  } else
    temp_d += 1.0;
}
```

Figure 2: Lightweight Modular Staging (LMS): (a) Num implementation and example usage (while loop), (b) generated C code for gradient computation.

Code generation

```
def snippet(filename: Rep[String]): Rep[Unit] = {
  // read data
  val scanner = new Scanner(filename)

  // set up our RNN model with rather arbitrarily-chosen hyperparameters
  // input to hidden
  val Wxh = TensorR.Tensor(Vector.randn(vocab_size, hidden_size, 0.01))
  // hidden to hidden
  val Whh = TensorR.Tensor(Vector.randn(hidden_size, hidden_size, 0.01))
  // hidden to output
  val Why = TensorR.Tensor(WhVector.randn(hidden_size, vocab_size, 0.01))

  val bh  = TensorR.Tensor(Vector.zeros(hidden_size))
  val by  = TensorR.Tensor(Vector.zeros(vocab_size))
  val hprev = TensorR.Tensor(Vector.zeros(hidden_size))
  val hnext = Vector.zeros_like(hprev)  // store the backward gradient here

  def lossFun(inputs: Rep[Array[Int]], targets: Rep[Array[Int]]) = {
    // loop through each character of the input
    val (f_loss, f_hs) = LOOPCCM((loss, hprev))(inputs.length){i => t =>
      val x = Vector.zeros(vocab_size)  // encode in 1-of-k representation
      x.data(inputs(i)) = 1
      val xs = TensorR.Tensor(x)
      val y = Vector.zeros(vocab_size)
      y.data(targets(i)) = 1
      val ys = TensorR.Tensor(y)

      // construct the loss function with auto-diff
      // hidden state
      val hs = ((Wxh dot xs) + (Whh dot t(1)) + bh).tanh()
      // new hidden state
      val es = (Why.dot(hs) + by).exp()
      // use unnormalized prob to compute normalize prob
      val ps = es / es.sum()
      // loss is updated by original loss t(0) and additional loss
      val newloss = t(0) - (ps dot ys).log()
      (newloss, hs) // new loss and new hidden state for next iteration
    }
    hnext.update(f_hs) // update the hidden state with the result from LOOP
    f_loss             // return the final loss
  }

  // Our RNN training code
  for (n <- (0 until NUM_LOOP): Rep[Range]) {
    // get a random sampling slice of the article
    val (input, target) = random_sampling(n)

    // forward input through the network and compute gradient (Auto-diff)
    val loss = gradR_loss(lossFun(input, target))(Vector.zeros(1))._1

    // Reverse-mode auto diff is done at the same time as forward computing.
    hprev.update(hnext, learning_rate)
  }
}
```

Figure 3: A Vanilla RNN in our DL framework

To evaluate our prototype, we trained a minimal character-level language model with a Vanilla Recurrent Neural Network and compare it to a Numpy implementation from Andrej Karpathy’s Github. Left hand side is the RNN model written in our simple deep learning framework. This code will generate a ~1000 line Cpp program to actually execute the model training task.

```
// ...header files

void Snippet();
int main(int argc, char *argv[])
{
  Snippet();
  return 0;
}

/*****************************************
Emitting C Generated Code
*****************************************/

void Snippet() {
  // read file
  int32_t x1 = open("input.txt",0);

  // allocate memory for Tensors
  Tensors = malloc()

  // start training model
  double final_loss;
  Tensor final_state;
  for(int x2=0; x2 < NUM_LOOP; x2++) {
    // allocate memory for necessary temp variables
    mem = malloc()

    // both the forward propgation and reverse-mode AD
    // are done in one recursive function call
    // each recusive call is a recurrence.
    function<pair<Tensor>> recursive_fun = [&](pair<Tensor>) {
      // when we performe forward prop, reverse-mode AD
      // is done at the same time
      forward_prop();
      reverse_mode_AD();
      // update hidden state and loss value in Tensor
      pair[0] = new_loss;
      pair[1] = new_state;
      if (not the last char in slice) recursive_fun(tensor_pair);
      else {
        // update final loss and state
        final_loss = pair[0];
        final_state = pair[1];
      }

      // free memory. Memory leakage is prevented by
      // Delimited Continuations
      free(mem)
    }

    // update parameter with gradients computed above
    para_update(learning_rate, final_state)

    free(Tensors)
  }

  // traning done!
}
```

Figure 4: The brief structure of generated Cpp program (~1000 lines)

Evaluation

- We implemented a minimal characterizer-level language model with Vanilla RNN in our lightweight deep learning framework and experimented it with an arbitrary article from Internet as input. By comparing it to a Numpy implementation, we see that our implementation works well on model training and runs faster than the Numpy one.

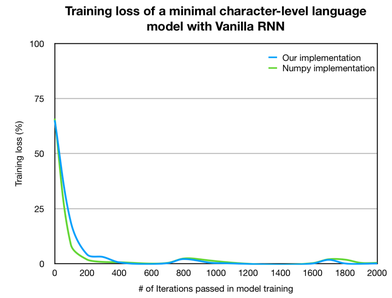


Figure 5: Training loss of a language model with an article input

	Numpy Implementation	Our Implementation
Exec time for 2000 iterations of model training (seconds)	2.816	1.120

Table 1: The run time of a minimal character-level language model with Vanilla RNN

Contribution

In this work, our main contribution includes:

1. We adapted Reverse-mode Automatic-differentiation to Delimited Continuations.
2. We demonstrated how MSP can be used to compile away the explicit auxiliary data structures used in popular deep learning frameworks.
3. We presented a well-integrated easy-to-extend deep learning framework in Scala which combines two orthogonal ideas firmly rooted in programming languages research.

We also notice that the hierarchy of deep learning frameworks may lose optimization potentials among each level and Partial Evaluation can help solve those perceived drawbacks.

Next steps & Ongoing work

- Of course we are not complacent about “beating” Vanilla RNN in Numpy. PyTorch embeds GPU acceleration and TensorFlow easily distributes large-scale model training to multiple processes or devices. We want to make our framework competitive to popular frameworks.
- XLA is a DSL JIT compiler for TensorFlow that optimizes operator computation. XLA takes the subgraph from TensorFlow and fuses multiple operators into a small number of compiled kernels. We want to adopt some similar optimization techniques in our framework to generate target code of better performance.
- The multi-level hierarchy of TensorFlow makes it difficult to debug while PyTorch makes full use of Python traceback facility. We believe that the good integration of our framework and host language can help us gain some deep properties. For example, we may want to apply state-of-the-art verification techniques on verifying the model we build is equivalent to what it’s supposed to be.

Huawei Midwest Research Summit 2018

Presented at Huawei Midwest Research Summit, March 8, 2018