Deep Learning in Rust: baby steps

Meta Edit 2/2/2016: I wrote this post when deeplearn-rs was only a week old. This is more of a journal post where I reflect on things. To see the latest deeplearn-rs, the readme on the github repo is probably the best place to look.

Last semester[1] I flailed around trying to get into deep learning. I felt kind of guilty for jumping onto the hype train[2]. Deep Learning is a guilty pleasure. I went through tutorials to relearn error backpropagation. Nowadays it's referred to as gradient flow too[3]. This course in particular helped me a lot.

After gaining a concrete understanding of the mechanics of deep learning, I wanted to experiment with real data and build real networks. I read lots of papers, but struggled to get even a few lines of code out. This was frustrating, because I'm the type of guy that *churns* code out. I was also kind of burning myself out, because my main project at the time was a ZFS driver for Redox. Dividing my brain between the behemoth that is ZFS and the still-young field of Deep Learning was exhausting and unproductive. So I shelved Deep Learning for a bit.

I was excited for winter break. I told myself I was going to *get stuff done*. I got some things done. I merged a massive PR for ZFS on Redox that I had been working on for about a month. My friend and I made the beginnings of a crappy stop motion fighting game. I played around with Tensorflow, went through some of their tutorials, and extended the language model tutorial code, but I wanted more. I struggled because I've never designed and carried out a full deep learning experiment with a modern framework. Nothing truly satisfying got done until the very end of winter break.

I found a nice compromise in a project that involved both deep learning and systems programming—a deep learning framework in Rust that uses OpenCL to do all the heavy lifting. Luckily, I had already written an OpenCL matrix math library[4]!

deeplearn-rs

I started hacking on this a week ago, and I'm pretty satisfied with how far it's come. You can construct and (manually) train a deep computation graph, and it all runs on the GPU via OpenCL. Take a gander:

```
extern crate deeplearn;
extern crate matrix;

use deeplearn::Graph;
use deeplearn::op::{MatMul, Relu};

fn main() {
    let ctx = matrix::Context::new();

    // Setup the graph
    let mut graph = Graph::new();
    let a = graph.add_variable(&ctx, (1, 2));
    // Layer 1
    let l1_w = graph.add_variable(&ctx, (2, 3));
    let l1_mat_mul = graph.add_node(&ctx,
                                    Box::new(MatMul::new(&ctx
                                    vec![a, l1_w],
                                    &[(1, 3)]);
    let l1_mat_mul_out = l1_mat_mul.get(&graph).outputs[0];
    let l1_relu = graph.add_node(&ctx,
                                 Box::new(Relu::new(&ctx, (1,
                                 vec![l1_mat_mul_out],
                                 &[(1, 3)]);
    let l1_relu_out = l1_relu.get(&graph).outputs[0];
    // Layer 2
    let l2_w = graph.add_variable(&ctx, (3, 1));
    let l2_mat_mul = graph.add_node(&ctx,
                                    Box::new(MatMul::new(&ctx
                                    vec![l1_relu_out, l2_w],
                                    &[(1, 1)]);
    let l2_mat_mul_out = l2_mat_mul.get(&graph).outputs[0];
    let l2_relu = graph.add_node(&ctx,
                                 Box::new(Relu::new(&ctx, (1,
                                 vec![l2_mat_mul_out],
                                 &[(1, 1)]);
    let l2_relu_d = graph.add_gradient(&ctx, l2_relu, 0);

    // Send some input data
    let a_cpu = matrix::Matrix::from_vec(1, 2, vec![1.0, -0.3
    let l1_w_cpu = matrix::Matrix::from_vec(2, 3, vec![0.5, 0
                                                       0.6, 0
    let l2_w_cpu = matrix::Matrix::from_vec(3, 1, vec![0.5, 0
    let l2_relu_d_cpu = matrix::Matrix::from_vec(1, 1, vec![1
    a.get(&graph).set(&ctx, &a_cpu);
    l1_w.get(&graph).set(&ctx, &l1_w_cpu);
    l2_w.get(&graph).set(&ctx, &l2_w_cpu);
```

```
47      l2_relu_d.get(&graph).set(&ctx, &l2_relu_d_cpu);
48
49      // Run the network
50      graph.run(&ctx);
51      let out = l2_relu.get(&graph).outputs[0].get(&graph).get(
52      let l1_w_d = graph.get_input_gradient(l1_w).unwrap().get(
53      println!("out = {:?}", out);
54      println!("l1_w_d = {:?}", l1_w_d);
55  }
```

deep_learn_xor.rs hosted with ❤ by GitHub view raw

Not as easy on the eyes as Theano or Keras or Tensorflow, I know. It's hard to beat python in the readability department[5], though. Anyway, this is lowest layer. I intend to write a `GraphBuilder` that makes constructing these networks a little more cushy and a little less error prone.

I'll just throw the deeplearn-rs github here.

If you don't know about deep learning and computation graphs, I'll try to elaborate. It turns out that artificial neural networks can be represented in a more general framework—computation graphs. Conveniently, artificial neural networks can even be represented by much more compact (in terms of node and edge count) graphs of matrix operations. For example, fully connected layers use matrix multiplication to multiply the weights and inputs and sum the products for each node. For each layer, all the inputs are represented as one matrix, and all the weights are represented as another. So convenient! Praise the math gods! It turns out that this makes it easy for us to implement super fast deep neural networks by running our (highly parallelizable) matrix operations on (highly parallel) GPUs.

Internals

Rust has strict ownership and borrowing rules. To make a long story short, you can't just dole out pointers to things like you can in most other languages[6]. So, instead I use indices into arrays. This way I don't have to deal with all of the headaches brought by ubiquitous use of `Rc<RefCell<T>>`[7]. Let's see what a node looks like:

```rust
pub struct Node {
    pub inputs: Vec<VarIndex>,
    pub outputs: Vec<VarIndex>,
    pub in_grad: Vec<VarIndex>, // gradients on inputs
    pub out_grad: Vec<OutGrad>, // gradients on outputs
}
```

deep_learn_node.rs hosted with ♥ by GitHub view raw

Every matrix[8] in a computation graph is represented as a variable. Variables are accessed using a corresponding `VarIndex`. All variables are stored in the graph's `VarStore`. VarStore uses `RefCell` to hide variable mutability so that multiple variables can be borrowed simultaneously, regardless of mutability. There is never a reason to borrow the same variable more than once at a time for our purposes.

Note the `out_grad` field is a list of `OutGrad`. OutGrad is a sort of CoW (Copy on Write) VarIndex to a gradient variable. When calculating gradients during the backward pass of the graph, if a variable is forked, (if a variable is used as input to multiple operations) there will be multiple gradients on the variable. If there are multiple gradients, they need to be summed to produce the final gradient, and a variable must be made to store that sum. If there is only one gradient, we can just use that gradient variable. OutGrad will only allocate a new variable if it needs to.

Let's look at the code for `Graph`, which ties everything together.

```
pub struct Graph {
    nodes: Vec<Node>,
    node_ops: Vec<Box<Operation>>,
    pub var_store: VarStore,
    out_var_map: HashMap<VarIndex, (NodeIndex, usize)>, // M
    // Gradients on variables that are inputs to the graph -
    in_var_grad: HashMap<VarIndex, OutGrad>,
}

impl Graph {
    pub fn new() -> Self {
        Graph {
            nodes: vec![],
            node_ops: vec![],
            var_store: VarStore::new(),
            out_var_map: HashMap::new(),
            in_var_grad: HashMap::new(),
        }
    }

    pub fn add_node(&mut self,
                    ctx: &matrix::Context,
                    op: Box<Operation>,
                    inputs: Vec<VarIndex>,
                    out_shapes: &[(u64, u64)])
                    -> NodeIndex {
        let node_index = NodeIndex(self.nodes.len());

        // Create output variables
        let mut outputs = vec![];
        for (i, &(rows, cols)) in out_shapes.iter().enumerat
            let var_index = self.var_store.add(ClMatrix::new
            outputs.push(var_index);
            self.out_var_map.insert(var_index, (node_index,
        }
        // Create input gradient variables and set up gradie
        let mut in_grad = vec![];
        for input in &inputs {
            // Create input gradient variables
            let (rows, cols) = (input.get(self).rows(), inpu
            let var_index = self.var_store.add(ClMatrix::new
            in_grad.push(var_index);

            // Set up gradient back flow
            match self.out_var_map.get(input).map(|x| *x) {
                Some((in_node, out_index)) => {
```

```
47                  self.nodes[in_node.0].out_grad[out_index
48              },
49              None => {
50                  // This input doesn't come from a node's
51                  self.in_var_grad.get_mut(input).unwrap()
52                      .fork(ctx, &mut self.var_store, var_
53              },
54          }
55      }
56      // Create the node
57      self.nodes.push(Node { inputs: inputs,
58                             outputs: outputs,
59                             in_grad: in_grad,
60                             out_grad: vec![OutGrad::new()
61      // Add the corresponding node op
62      self.node_ops.push(op);
63      node_index
64  }
65
66  pub fn add_variable(&mut self, ctx: &matrix::Context, sh
67      let v = self.var_store.add(ClMatrix::new(ctx, shape.
68      self.in_var_grad.insert(v, OutGrad::new());
69      v
70  }
71
72  pub fn get_input_gradient<'a>(&'a self, v: VarIndex) -> (
73      self.in_var_grad.get(&v).and_then(|x| x.try_gradient
74  }
75
76  pub fn add_gradient(&mut self, ctx: &matrix::Context, n:
77      let (rows, cols) = {
78          let grad = n.get(self).outputs[out_index];
79          let grad = grad.get(self);
80          (grad.rows(), grad.columns())
81      };
82      let v = self.var_store.add(ClMatrix::new(ctx, rows a
83      self.nodes[n.0].out_grad[out_index].fork(ctx, &mut s
84      v
85  }
86
87  pub fn run(&mut self, ctx: &matrix::Context) {
88      // Forward pass
89      //
90      // NOTE: We just execute the nodes in order. We can
91      // built. When a user wants to add a node, he/she mu
92      // any dependencies must already be added before the
```

```
        // any dependencies must already be added before the
        // assert that all dependents come after their depen
        for (node, op) in self.nodes.iter_mut().zip(&mut sel
            op.forward(ctx, &mut self.var_store, node);
        }

        // Backward pass
        for (node, op) in self.nodes.iter_mut().rev().zip(se
            // Sum the gradients on each output if there are
            for out_grad in &node.out_grad {
                out_grad.maybe_sum(ctx, &mut self.var_store)
            }
            op.backward(ctx, &mut self.var_store, node);
        }
    }
}
```

deep_learn_graph.rs hosted with ♥ by GitHub view raw

The most interesting function here is `Graph::add_node`. You give it an operation and a list of inputs and it'll handle the rest. It creates the output variables and creates a new gradient for each of the inputs.

Lastly, let's look at the implementation of matrix multiplication:

```
pub struct MatMul {
    a_t: ClMatrix<f32>,
    b_t: ClMatrix<f32>,
}

impl MatMul {
    pub fn new(ctx: &matrix::Context, a_shape: (u64, u64), b_
        MatMul {
            a_t: ClMatrix::new(ctx, a_shape.1 as usize, a_sha
            b_t: ClMatrix::new(ctx, b_shape.1 as usize, b_sha
        }
    }
}

impl Operation for MatMul {
    fn forward(&mut self, ctx: &matrix::Context, v: &mut VarS
        let a = &v.get(n.inputs[0]);
        let b = &v.get(n.inputs[1]);
        let c = &mut v.get_mut(n.outputs[0]);
        a.dot(ctx, b, c); // c = a*b
    }

    fn backward(&mut self, ctx: &matrix::Context, v: &mut Var
        let a = &v.get(n.inputs[0]);
        let b = &v.get(n.inputs[1]);
        let a_d = &mut v.get_mut(n.in_grad[0]);
        let b_d = &mut v.get_mut(n.in_grad[1]);
        let g = &v.get(n.out_grad[0].gradient());

        // Derivative with respect to first input
        // a_d = g*b_t
        b.transpose(ctx, &mut self.b_t);
        g.dot(ctx, &self.b_t, a_d);

        // Derivative with respect to second input
        // b_d = a_t*g
        a.transpose(ctx, &mut self.a_t);
        self.a_t.dot(ctx, g, b_d);
    }
}
```

deep_learn_mat_mul.rs hosted with ♥ by GitHub view raw

The boilerplate variable fetching is a bummer.

Conclusion

Overall, I like the design so far. The purpose of deeplearn-rs is to provide a framework to build trainable matrix computation graphs that are configurable at runtime. If configuration at runtime weren't a requirement, I'd almost rather just code the matrix operations and the backward gradient flow by hand.

Moving forward, some things I'd like to do:

- Implement some loss functions
- Train a fuzzy xor network
- Automate the training process, and provide several different trainers such as vanilla SGD, ada grad, etc.

[1] Fall 2015
[2] But then less guilty because I was into artificial neural networks back in high school, and deep learning is a generalization of those ideas
[3] Error backpropagation never really rolled off the tongue
[4] The matrix math library also got some sweet upgrades
[5] Nigh impossible
[6] In safe Rust you can't, but in unsafe Rust, anything is possible…
[7] `Rc<T>` is a reference counted pointer, but it demands that the inner type be immutable. `RefCell` hides mutability at compile time and ensures that all mutable references are exclusive at run time. Any number of immutable borrows from a value can exist at any given time, but a mutable borrow must be the only borrow for the duration of its lifetime.
[8] Whether an input, a weight, a gradient, etc.