



Knight Foundation
School of Computing
and Information Sciences



Deep Learning Framework in Rust

Lazaro Hurtado, Ngang Bah, Julio Rego, Roberto Martinez



FLORIDA
INTERNATIONAL
UNIVERSITY



Purpose

Deep learning is a rapidly evolving field that has seen significant advancements in recent years, resulting in the development of powerful algorithms for tasks such as image recognition, natural language processing, and speech recognition. However, most existing deep learning frameworks are written in languages like Python and C++, which can limit their performance and scalability.

Our goal with this project is not only to develop a fast and efficient deep learning framework but also to contribute to the larger Rust and machine learning communities. In particular, we aim to contribute to the "AreWeLearningYet" project, which is a community-driven effort to create a comprehensive list of Rust machine learning libraries, frameworks, and tools. By developing a deep learning framework in Rust, we hope to expand the ecosystem of machine learning tools available in Rust and contribute to its growth and adoption.



Features

Our Deep Learning framework in Rust offers a flexible computation graph, support for various activation and loss functions, several popular optimizers, and a range of modules including linear and convolutional layers. With an intuitive and simple API that PyTorch developers may find familiar, our framework provides a reliable, efficient, and high-performance tool for building and deploying deep learning models

```
impl<D> CrossEntropy<D, D>
where
    D: Dimension + RemoveAxis,
{
    fn with_class_probabilities(
        probabilities: &Tensor<D>,
        class_probabilities: &Tensor<D>,
    ) -> Tensor<D> { ...

    fn normalized_with_prob<E: Dimension>(
        probabilities: &Tensor<E>,
        class_probabilities: &Tensor<E>,
    ) -> Value { ...
} impl CrossEntropy<D, D>
```

CrossEntropy Loss function

```
pub struct SGD {
    pub params: Vec<Value>,
    pub lr: Value,
    pub momentum: f64,
    pub dampening: f64,
    pub weight_decay: f64,
    pub maximize: bool,
    pub cache: SGDCache,
}
```

SGD Optimizer Struct

```
impl Linear {
    pub fn new(name: impl ToString, nin: usize, nout: usize) -> Self {
        let name = name.to_string();
        let weights =
            Tensor::from_shape_simple_fn(
                shape: (nin, nout), f: || GlorotUniform.sample(fanning: [nin, nout])
            );
        let biases = Tensor::from_shape_simple_fn(shape: nout, f: Value::zero);

        Linear {
            name,
            weights,
            biases,
        }
    }
}
```

Linear layer



More Features...

```
impl<D> Activation<D> for Sigmoid
where
  D: Dimension,
{
  fn activate(&self, unactivated: &Tensor<D>) -> Tensor<D> {
    unactivated.mapv(|value| Value::one() / (Value::one() + (-&value).exp()))
  }
}
```

Sigmoid activation function

```
impl Schedule for StepLR {
  fn get_lr(&self, lr: f64, last_epoch: usize) -> f64 {
    if (last_epoch == 0) || (last_epoch % self.step_size != 0) {
      lr
    } else {
      lr * self.gamma
    }
  }

  fn get_closed_form_lr(&self, base_lr: f64, last_epoch: usize) -> f64 {
    let power = (last_epoch / self.step_size) as i32;

    base_lr * (self.gamma).powi(power)
  }
}
```

Step Learning Rate Scheduler

```
impl WeightInit for GlorotNormal {
  fn sample<F: Into<Fanning>>(&self, fanning: F) -> crate::Value {
    let Fanning(fan_in, fan_out) = fanning.into();
    let stdev = (2.0 / (fan_in as f64 + fan_out as f64)).sqrt();
    let normal = Normal::new(mean: 0.0, std_dev: stdev).unwrap();

    let mut rng = rand::thread_rng();

    let weight = normal.sample(&mut rng);
    val!(weight)
  }
}
```

Normal Glorot Weight Initializer



Shortcomings

Limited Scalability

In the early stages of our project, we opted to build our differentiation graph at the level of individual scalar values using a Value struct, hence the name “Micrograd.” While this approach is efficient for smaller models, it poses significant challenges when dealing with larger models. In hindsight, it would have been more efficient to construct our differentiation graph at the level of individual tensors, this would have resulted in a smaller graph size making the system much more scalable.



Conclusion

In conclusion, our Rust-based Deep Learning framework is a powerful tool that offers a range of features and capabilities for building and deploying complex machine learning models.

Moving forward, we plan to continue to expand the framework's capabilities by creating examples around popular models like Transformers for NLP and YOLO for computer visions. In addition, we plan to increase efficiency in the auto differentiation graph backward pass. With these enhancements, we believe that our Deep Learning framework will continue to be a valuable tool for developers working in the field of machine learning, and contribute to the continued growth and evolution of the Rust machine learning ecosystem