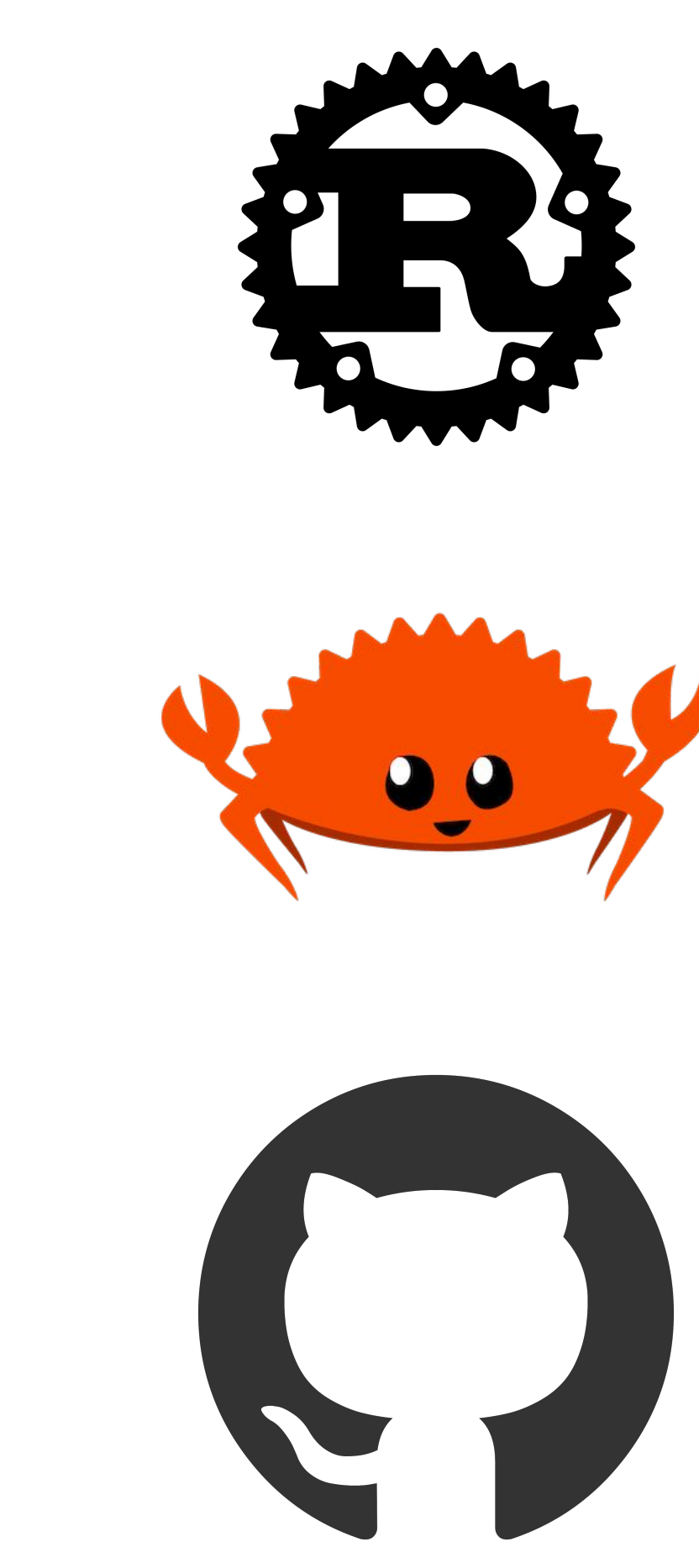


Student: Roberto Martinez
Instructor/Faculty: Dr. Masoud Sadjadi, Florida International University



PROBLEM

Our project aims to tackle the challenge of creating deep learning frameworks that are both efficient and scalable enough to handle large datasets and models. Specifically, my work was dedicated to creating fundamental functions necessary to optimize for overfitting and different probability methods. These included the sigmoid activation function and the dropout method.

```
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()))
    }
}
```

fig 1: Sigmoid Activation Function

CURRENT SYSTEM

We developed our own Deep Learning framework from the ground up, drawing inspiration from Andrej Karpathy's micrograd and optimizing it for Rust's capabilities. Our framework provides a versatile computation graph, allows for multiple activation functions, and has an intuitive API that should be easy for any Pytorch developer to adapt to.

```
#[test]
fn valid_softmax_activation() {
    let inputs = arr1(&VALUES_1D).mapv(|val| Value::from(val));
    let softmaxed = Activation::Softmax(0).forward(&inputs).into_raw_vec();
    let outputs = softmaxed.iter().map(|output| output.value());

    let actuals = [0.18047813, 0.08440353, 0.01773622, 0.71738213];

    for (output, actual) in outputs.zip(actuals) {
        assert_abs_diff_eq!(output, actual, epsilon = 1e-6);
    }
}
```

fig 2: Sigmoid Activation Function Tests

REQUIREMENTS

- Provide developers with core features, such as:
- Activation functions
 - Linear, Pooling, and Convolutional layers
 - High readability and code hygiene
 - Loss functions
 - Optimized for overfitting

CONTRIBUTIONS

- Implemented Sigmoid activation function, often used in binary classification problems to produce a probability value that an input belongs to a particular class. [fig 1]
- Implemented the Dropout function to prevent overfitting of our data. It works by randomly dropping out (setting to zero) a certain percentage of the neurons in a layer during each training iteration. [fig 3]
- Created robust testing suites for each of the implementation to mitigate misleading outputs. [fig 2]

SYSTEM DESIGN

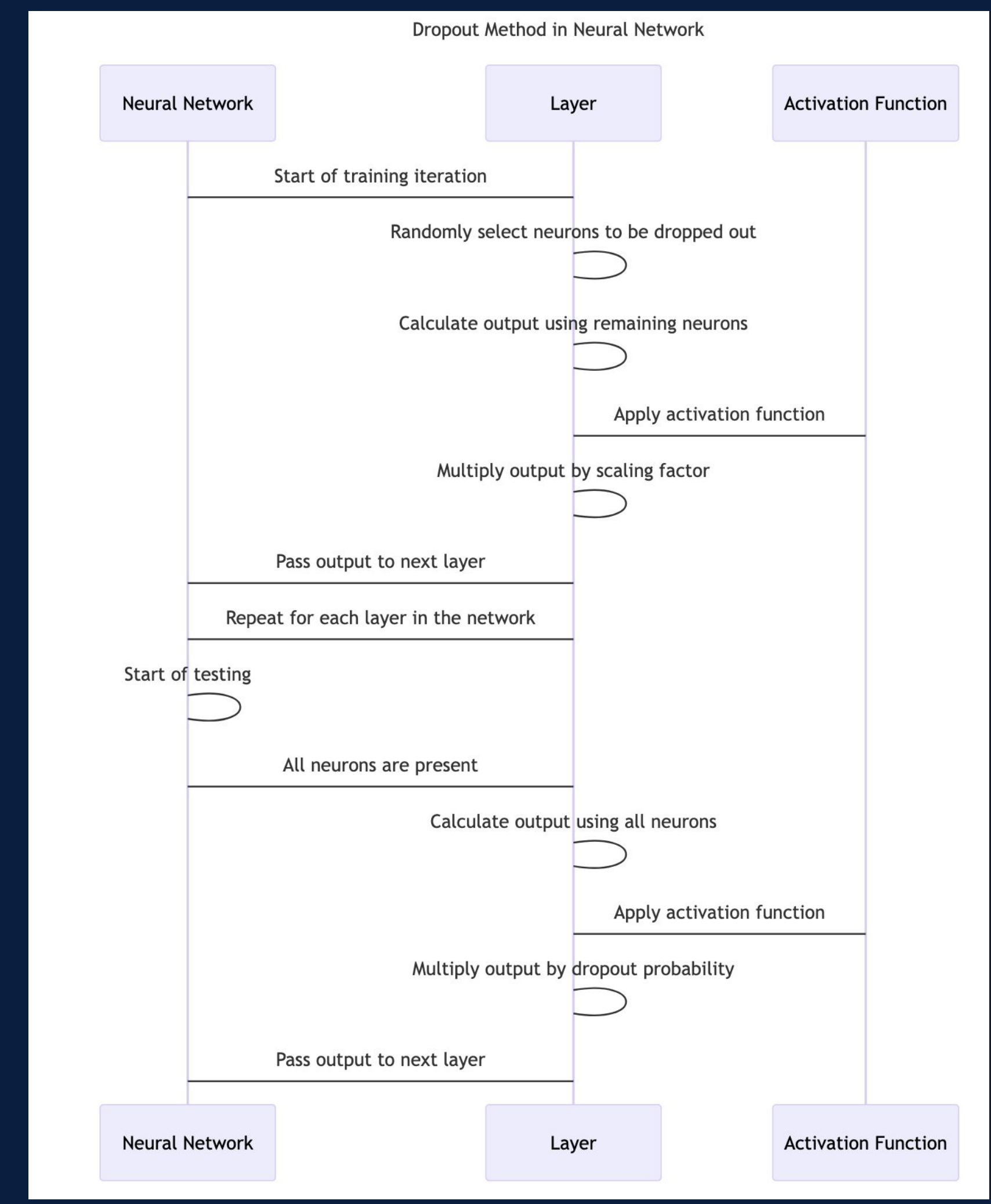


fig 5: How a user would call their model

```
impl Dropout {
    pub fn new(p: f64, seed: Option<u64>) -> Self {
        Dropout {
            p,
            mask: None,
            training: true,
            seed,
            rng: None,
        }
    }
}
```

Fig 3: Dropout Parameters

SUMMARY

Our goal in creating our framework from the ground up is to provide developers with a user-friendly, high-performance tool that can be used to build and deploy robust features in Rust [fig 4]. Thanks to our framework's optimized performance and intuitive design, it is an ideal choice for those who wish to explore new machine learning models and drive innovation in the field of deep learning.

REFERENCES

- Andrej's micrograd: <https://github.com/karpathy/micrograd>
- Pytorch documentation: <https://pytorch.org/docs/stable/nn.html>
- Micrograd_rs: <https://github.com/LazaroHurtado/micrograd-rs>