Machine Learning project report

Implementing a Neural Networks & data preprocessing framework from scratch in Rust in order to solve a regression problem

Anicet Nougaret

Abstract

I implemented a Neural Networks (NNs) and data preprocessing mini-framework in Rust using dataframes, linear algebra and data serialization libraries as my only dependencies. My goal was to create a data preprocessing pipeline and a NN model for doing performant supervised learning on the King County House dataset. At first I struggled with the limited features of my framework and my limited intuition for NNs based approaches. Trying around 100 small models with varying success, getting stuck trying irelevant techniques for small models such as Dropouts, too naively initializing the learnable parameters and not preprocessing the data enough. But after reading more related papers, trying many hyperparameters, implementing proper data preprocessing such as squared feature extraction, normalization and outliers filtering, setting up proper k-fold training, model benchmarking and implementing performant techniques such as Adam optimization, ReLU activation and Glorot initialization, I was able to build a better intuition and scale my models up by using more data features. Using my framework configured with almost the same defaults as the Keras Python library, I finally recreated an existing performant Python Keras workflow for King County houses price inference, and obtained a model with $R^2 = 0.83$. Then, I stabilized the framework's API in an easier to use, documented package and shared the library to the Rust community. I obtained great feedback and downloads on the Rust libraries repository. I look forward to learning more about NNs and Machine Learning, and growing my framework over time.

This report will not include source-code snippets throughout for readability reasons, but you will find relevant listings at the end of the report. Here is a link to the project's source code and here is a link to its online documentation.

Contents

1. Introducing NNs: Learning the xor function

Learning the xor function is a basic first step when implementing NNs. It helps checking whether the network learns anything at all from the observations. Doing this early-on helped me implement the basic features, learn a few things about how NNs work and make sure I had a solid basis for the next steps.

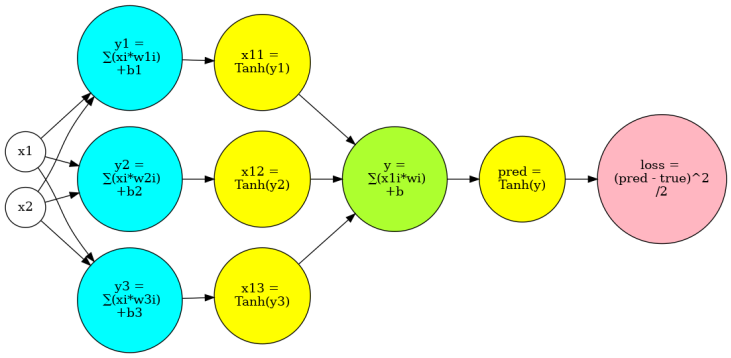

Figure 1: Network used to learn the xor function.

I used a rather standard model for such a problem, consisting of 2 dense layers with 2 input neurons, 3 hidden neurons, 1 output neuron, and $\tanh$ activations for the hidden and output neurons.

1. a. SGD/MSE: How and what a Neural Network learns

The network, fed with all the possible *(input, output)* values for the xor function, repeatedly for 1000 epochs, first executes a *forward pass*, which for each layer from first to last, computes an output by doing the sum of its inputs multiplied by its weights plus its biases. Since the output has a linear relationship to the input, which may not represent how the reality works, we pass it through a non-linear *activation function*, $\tanh(x)$ in our case. Then it calculates the *loss* of the activated prediction relative to the true value, using a *Mean Squared Errors (MSE) loss function*, which is a common loss function helping to converge towards both a low variance and bias model.

Then, in order to converge towards a local minimum of the loss function, it executes a *backward pass* using the *Stochastic Gradient Descent (SGD)* algorithm. This algorithm computes for each layer from last to first the gradient of the loss function with respect to each *learnable parameter* (e.g. weights and biases), and updates the parameters in the opposite direction of the gradient, multiplied by a *learning rate* hyperparameter. It then passes the gradient of its input (e.g. the previous layer's output) with respect to the loss function, so that the previous layers can repeat the same process for themselves. This step is called *backpropagation*.

After the last epoch, hopefully, the loss converged to a local minimum and predictions start looking quite good.

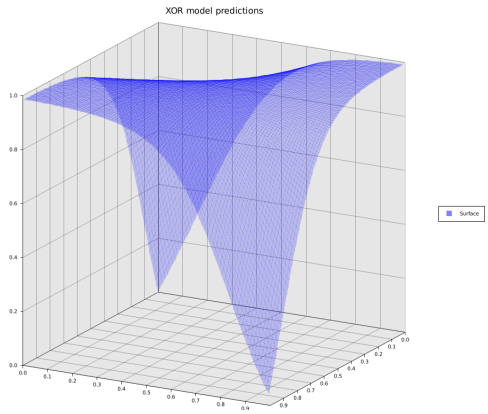

Figure 2: Predictions for inputs ranging from $(0.0, 0.0)$ to $(0.1, 0.1)$

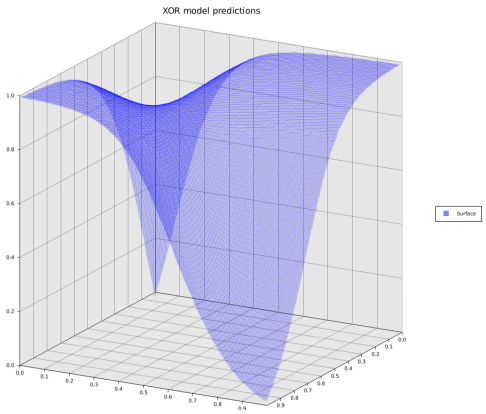

Figure 3: Not all trainings lead to the same predictions. This is another local minima found, hence the new shape

As we can see, the NN which had weights and biases initialized at random in the $[0, 1]$ range, tried to generalize based on the limited observations it received, and therefore can predict what the *non-existant* value of xor would be for any input in the $[0, 1]$ range. But since the starting parameters are set at random, it converges to different minimas each time, which leads to different predictions.

1. b. Adding Learning Rate Decay and Dropouts

I then added *learning rate decay* which introduces two hyperparameters: r_0, the *initial learning rate* and d, the *decay rate*. Then, instead of using a constant learning rate during backpropagation, it updates r_{i+1}, the learning rate for each iteration $i + 1$ as:

$$r_{i+1} = \frac{r_i}{1 + d.\, i}$$

This helps the model converge more precisely as it gradually "slows its descent", enabling fast exploration at the beginning, but preventing over-shooting in the long run. Finding the right value is tricky, as too high values can lead to the model converging too slowly.

I also added *dropouts* which is a technique used to prevent *overfitting* by randomly droping nodes of each layer with a probability p during the forward pass (by setting their input or activation to 0), and adapting the backward pass accordingly. In a way it simulates learning from a different, simpler model at each epoch, building more varied and robust features. I think this helps the model generalize better for large problems with a lot of inputs of similar importance, inputs resistant to compression, dimensionality reduction, such as pictures. But here it drops too much information as it is a very small model with a high dependency on both inputs.

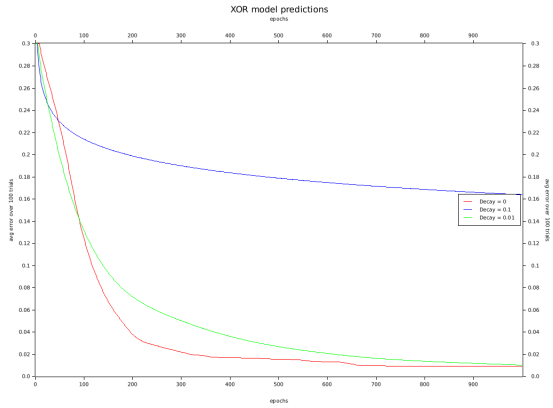

Figure 4: High decay (blue) learns too slowly

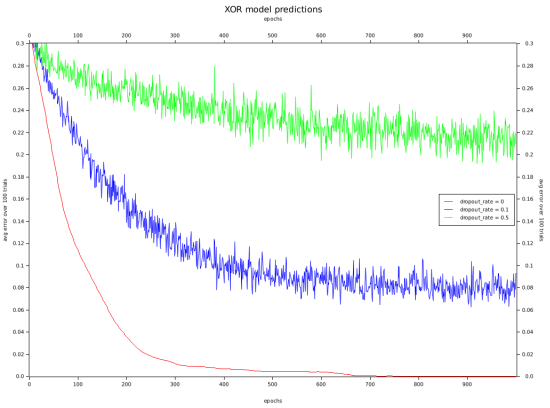

Figure 5: Dropout (blue & green) make simple models worse by droping too much information

I slowly started realizing how over-engineering with fancy features what should be a simple model for a simple problem wouldn't help.

2. Price inference on the King County Houses dataset

2. a. Normalization and trying initial hyperparameters

For problems dealing with real data, we can't just feed it and predict right away, as real-world data can have large range and skewed distributions which does not help the network learn.

So I got started with *feature scaling* by normalizing the data features using *min-max normalization*: scaling them on a range between the feature's min and max.

$$x' = \left(\frac{x - \min(x)}{\max(x) - \min(x)} \right)$$

Then I started building small models using a subset of features. I tried many hidden layers count, layers sizes, and hyperparameters, by increasing or decreasing them gradually and looking at whether the training loss would decrease faster or further.

At some point I was stuck with better models than at the beginning, but still a too high training loss. And a proportional distance between predicted and true values indicating predictions on average 20% to 40% off.

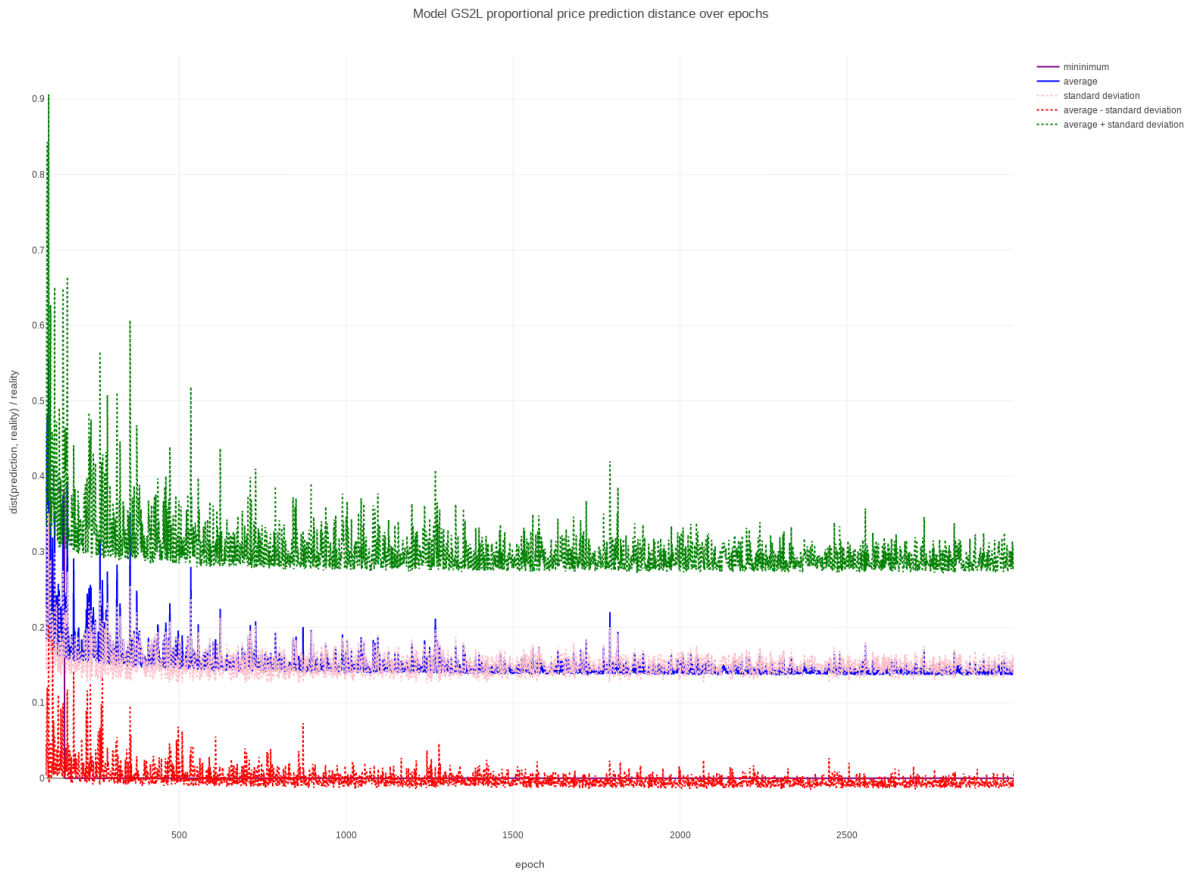

Figure 6: The $\pm 20\%$ average proportional distance, with $\pm 20\%$ standard deviation is not great. Is this better than random guessing?

2. b. K-folds cross validation and model debugging

I made it possible to write *JSON* specifications for the models so that I could easily load them, train them, compare them.

I also started using *k-folds* training, and saved all predictions made on each fold during the last epoch, so that I could finish my training with a prediction for each row of the dataset.

What I realized once I plotted all the predicted prices against the actual prices, is that the model only learned the value's statistical distributions and was guessing at random in that distribution, not being able to use the inputs to scale the output up or down.

Indeed, this is a possible local minimum of the loss function. If the model is not able to use the inputs to predict the values, at least it learns the distribution on the long run because it is the best thing it can do to minimize the MSE.

Figure 7: It only learned the distribution. The purple predicted values don't follow the true values in green. (log scaled prices)

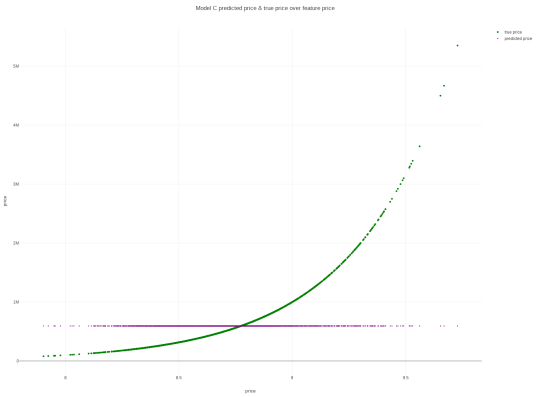

Figure 8: This model learned it even better, it now always predicts the exact same value that statistically makes sense.

After that, I got rid of the average proportional distance metric I was using so far, which proved not to reveal the whole picture. I started using the epoch average training loss and average testing loss metrics.

2. c. Specifying input features and preprocessing pipelines

I conjectured that the input features I had previously chosen (*grade*, *bedrooms count*, *condition*, and *yr_build*) were not the most significant ones and did not help the model predict. Although they are highly correlated with the price, they may not explain the price well in comparison to other unused features.

So I added the possibility to specify the features and their preprocessing (normalization, logarithmic scaling…) in the model's configuration file. I also implemented a cached and revertible data pipelining functionality, in order to make it easier to add features, preprocess them, and attach original unprocessed features to the predicted values.

2. d. Adding more features, getting better but unstable models

After adding more significant input features such as the house's *latitude*, *longitude* and *squared feets*, using a small model consisting of one 9 nodes hidden layer, using SGD with a constant learning rate I started getting better results:

Figure 9: The predicted prices follow the true prices much better.

As we can see at the top of the predicted values cloud there is a dense area of predicted values. This is because the data is divided by the number of folds, and each fold's model at the final epoch does the predictions on its own data. Therefore, it looks like one of the folds had issues learning and predicted a narrow range of values.

What I realized when training these models is that the models were very unstable, and from one fold to another we would get almost no decreasing loss and a model that only learned some distribution, or highly decreasing loss and good looking results. It was not an overfitting issue since faulty models had both high training and testing loss. It was looking like a converging issue, where some random parameters of the models randomly got it stuck during gradient descent.

Figure 10: The k models are unstable, some models learned skewed or narrow distributions

2. e. Adding mini-batches and momentum SGD

I added the possibility to train with *mini-batches*. Instead of training one row at a time, the model trains on a batch of rows at a time. This is much faster as it takes advantage of the linear algebra libraries by doing parallel computations on large matrices instead of single-thread computations on vectors. This was a difficult refactor as it had repercussions on the whole layers-related code.

I also added the possibility to train with *momentum SGD*. Like a ball rolling down a hill, the momentum helps the model get out of local minima and reach a better global minimum.

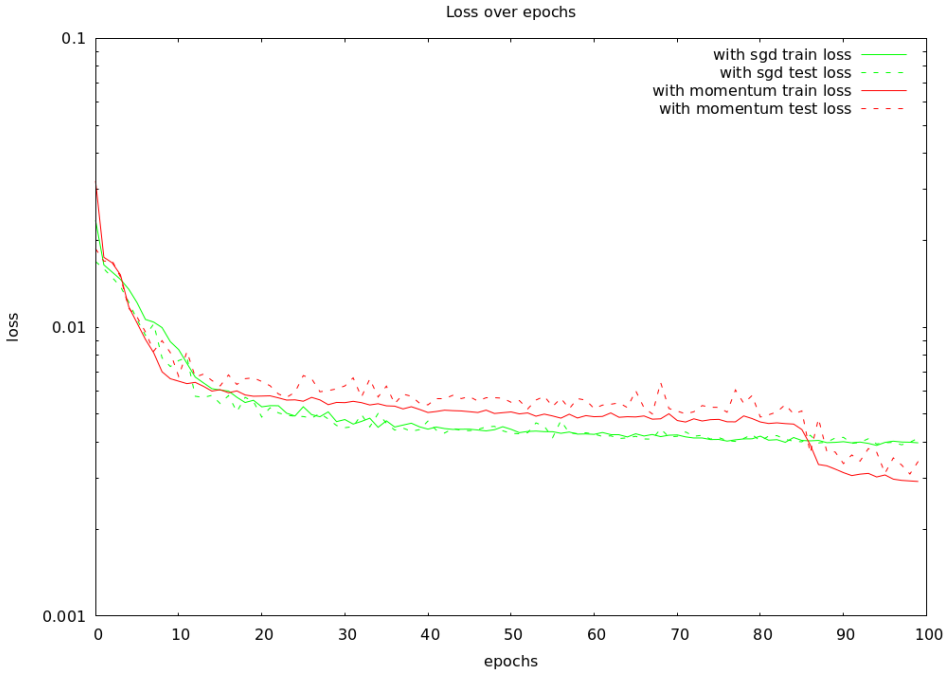

Figure 11: SGD vs Momentum on the final model. Momentum ends up finding a better minimum. Since epoch ~ 25 SGD floored at 0.004 training loss. Momentum had all its folds at 0.002 but had one badly performing fold at 0.016 which made it all worse on average. After 75 epochs thought this bad fold got unstuck. Thanks to the momentum?

2. f. ReLU, squared features, Glorot Uniform weights and 0 biases initialization

At first I was struggling to make the switch to ReLU activation as I had very unstable results using it with previous models.

But I was initializing the biases with a random value between 0 and 1 instead of just initializing them to 0. In my experiments it did not make the $\tanh$ activated model significantly less stable, but it made ReLU highly unstable.

I suspect that having negative biases at the beginning got the model stuck with ReLU returning always zero without any way for the gradient to help the bias to get back to a positive value.

Also, ReLU is linear so it does not learn non-linear relationships well. I added squares of the input features to the input data which helped it learn.

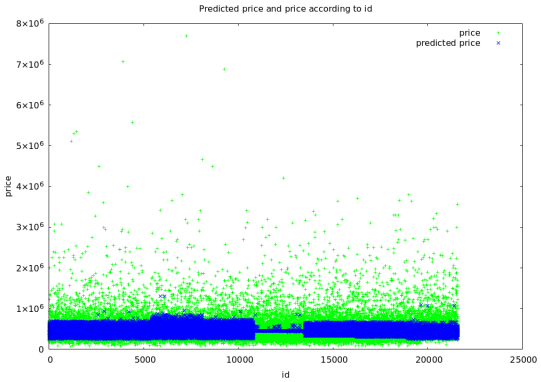

Figure 12: With ReLU with squared features and biases initialized at 0

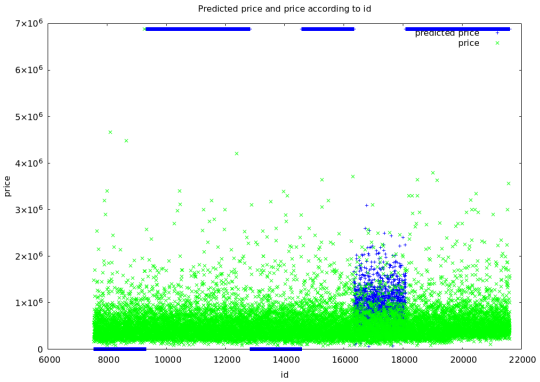

Figure 13: With ReLU with squared features and biases initialized in $[-1, 1]$

I also started implementing the *Glorot Uniform* weights initialization, since it was Keras' default weights initializer. I read the original paper, implemented it, but struggled to really see why it works. But it does improve the models' stability and performance, whether using ReLU or $\tanh$ activation.

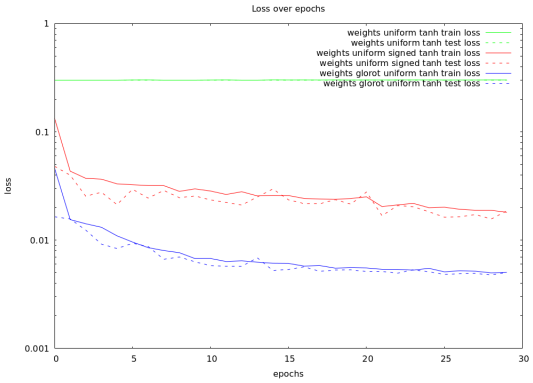

Figure 14: Comparing uniform, uniform signed and Glorot uniform weights initialization with $\tanh$. Glorot outperforms the other two.

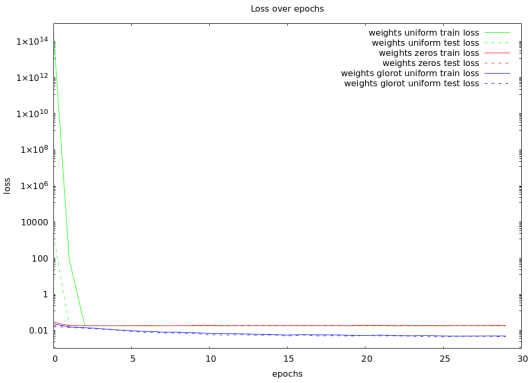

Figure 15: Comparing uniform, zeros and Glorot uniform weights initialization with ReLU. Glorot outperforms the other two.

The other great advantage of using ReLU is that it is a very fast activation function. And Glorot is the best initialization I found when using ReLU.

2. g. Log scaling and outliers filtering

One issue I had with the data was its skewed distribution which did not help the model. One fix to that issue was to log scale the features that had distributions skewed towards lower values, such as the *price* or the *squared feets* features.

Another issue with the data was the high number of *outliers*, e.g. extreme values that are too isolated for the model to learn anything from them. I filtered them out using *Tukey's fence method* which removes rows with one of their features' value above or below 1.5 times that feature's *interquartile range (IQC)*.

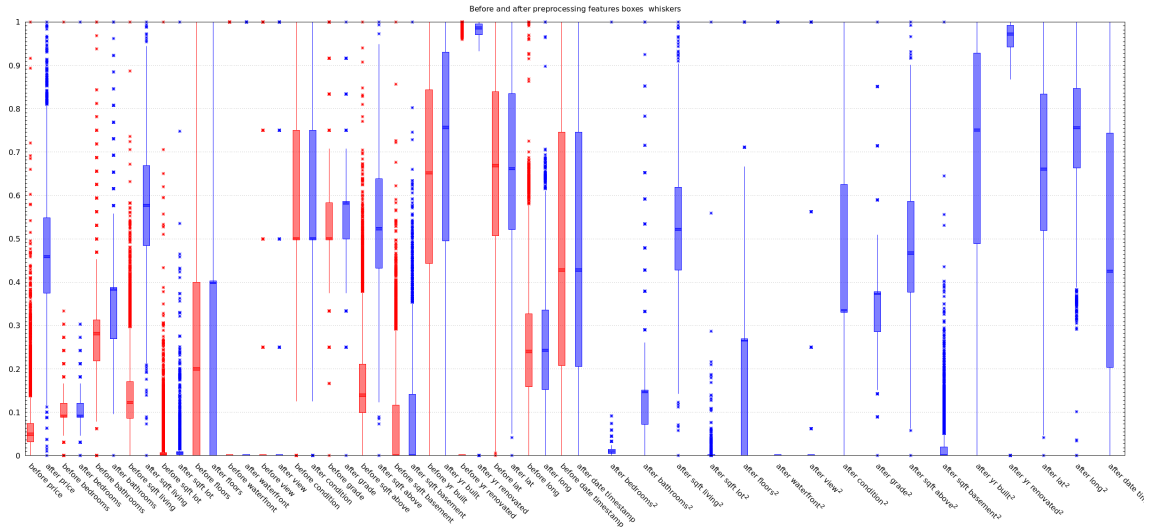

Figure 16: Box and whisker plots of the features' distributions before (red) and after(blue) the full preprocessing pipeline. The log scaled features are much more normally distributed. The outliers are also much less extreme.

I found out that filtering outliers makes the model faster by almost dividing the input rows by a factor of 3, but makes it slightly less accurate.

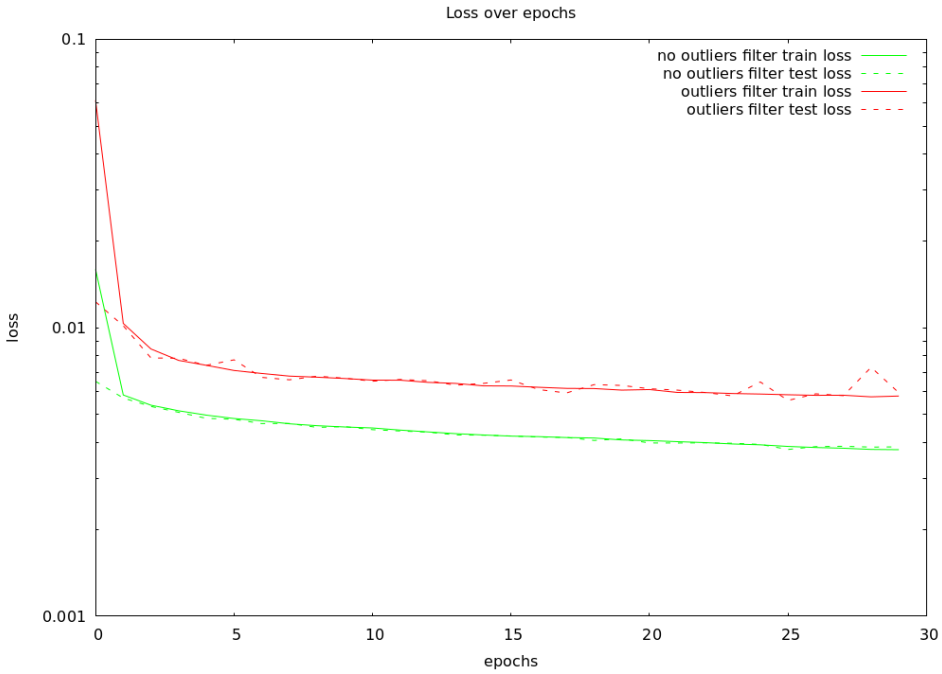

Figure 17: The red model was trained on ~ 750 rows, versus ~ 2625 for the red one, and only has an average of ~ 0.002 more loss.

2. h. Implementing Adam and comparing it to Momentum and SGD

The final step in having the same defaults as the Keras framework, was using the *Adam* optimizer instead of Momentum.

It uses the momentum and moving average of the gradient to update the parameters, allowing it to handle sparse gradients and noisy data, like ours, more efficiently.

I implemented Adam by following the algorithm found in the original paper. It introduced a few additional hyperparameters that I set to Keras' defaults.

With the King County house price regression problem it does not perform better than SGD nor Momentum with default parameters. It converges faster to a low loss during the 10 initial epochs, but it gets floored at an higher error than SGD or Momentum in the long run. Maybe with some hyperparameter fine-tuning it would outperform, but I haven't got time to try that yet.

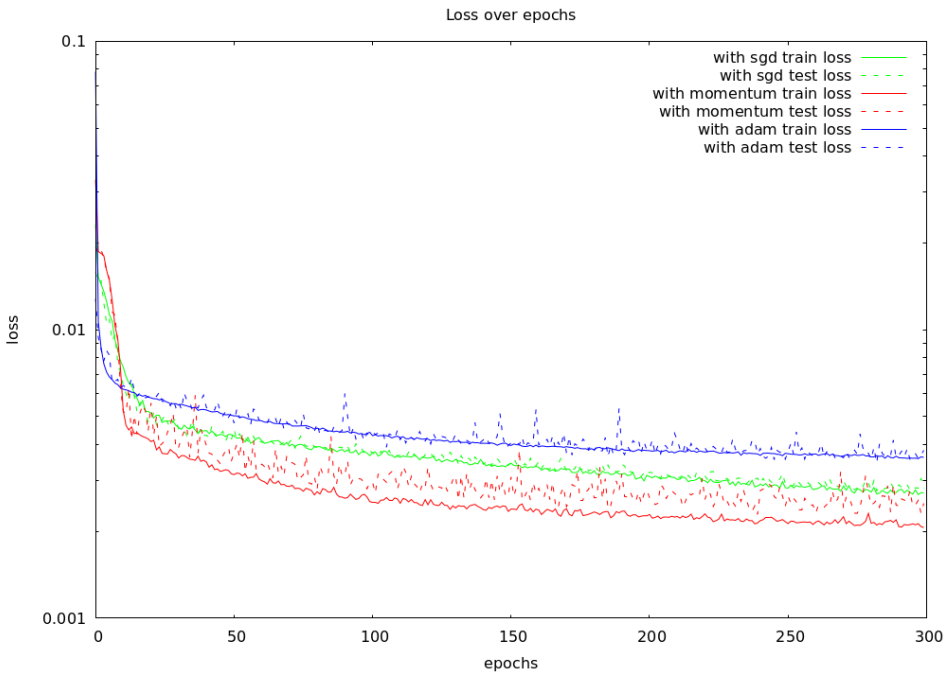

Figure 18: Adam, SGD and Momentum

2. i. Final results

After all these tweakings and experiments, I ended up with a model consisting of 8 hidden layers of as many neurons as inputs (as I figured out during my experiments that it was the best architecture for my data), using ReLU activation until the last layer which has linear activation, Glorot uniform weights initialization, and Momentum optimizer with a constant learning rate set to 0.001.

I trained it for 300 epochs and then computed the R^2 score for the model which is a standard regression quality metric (with scores from 0: worst, to 1: best) computed as:

$$R^2 = 1 - \frac{\sum\left(y_i - \hat{y}_i\right)}{\sum\left(y_i - \bar{y}\right)}$$

With y_i the i^{th} true price and $\hat{y}_i$ the i^{th} predicted price (computed at the final epoch of the fold in charge of the i^{th} row).

This model attains an $R^2 = 0.8304801$.

It is worse than the Python Keras workflow I got inspiration from at $R^2 = 0.88$, and would be even worse if I used the same defaults as it does since it uses Adam, which underperforms on my own case. But this can be explained by the fact that we don't do the exact same preprocessing pipeline. The original one is more advanced and would require more updates to my framework before I could do it the exact same.

Here are some charts of the predictions (in blue) and true prices (in green) according to interesting features:

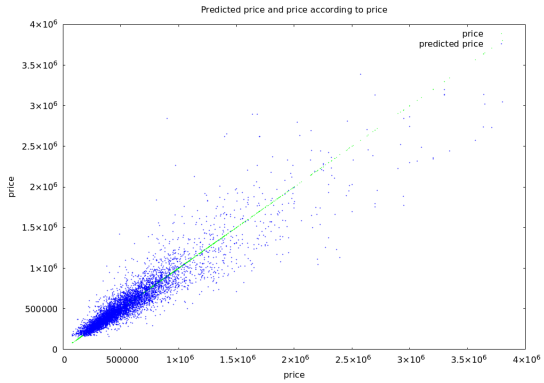

Figure 19: Predictions grow with actual value as expected. Higher prices seem to be harder to predict. Maybe they are too shallow.

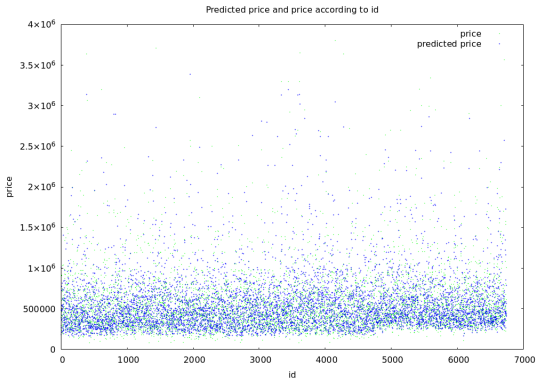

Figure 20: The unstable folds issue is far gone at that point

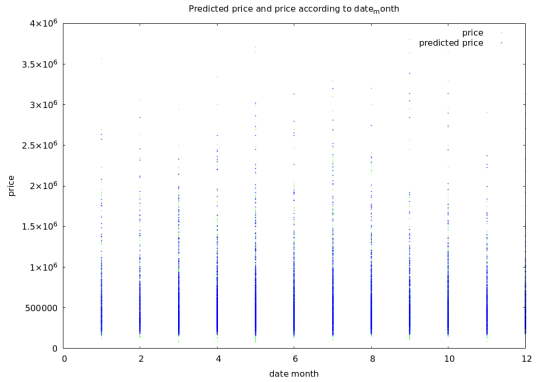

Figure 21: You don't want to buy your house in July it seems.

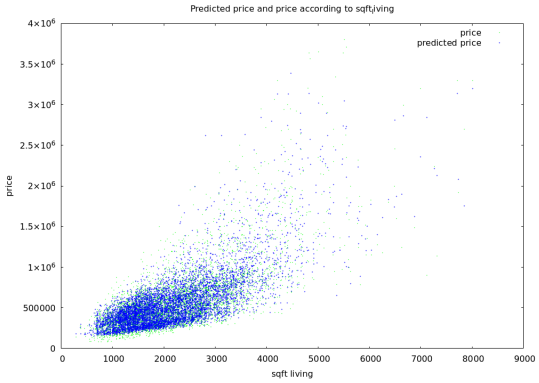

Figure 22: The living room's squared feets has a significant correlation with price. But is from enough to explain it.

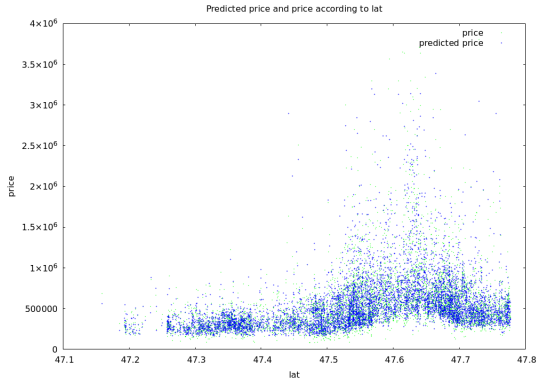

Figure 23: Latitude is by far the most impactful feature.

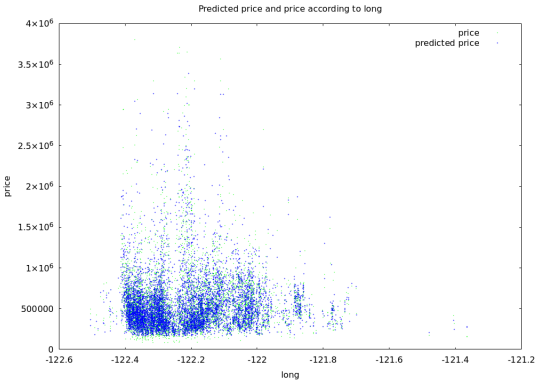

Figure 24: Longitude also has an impact

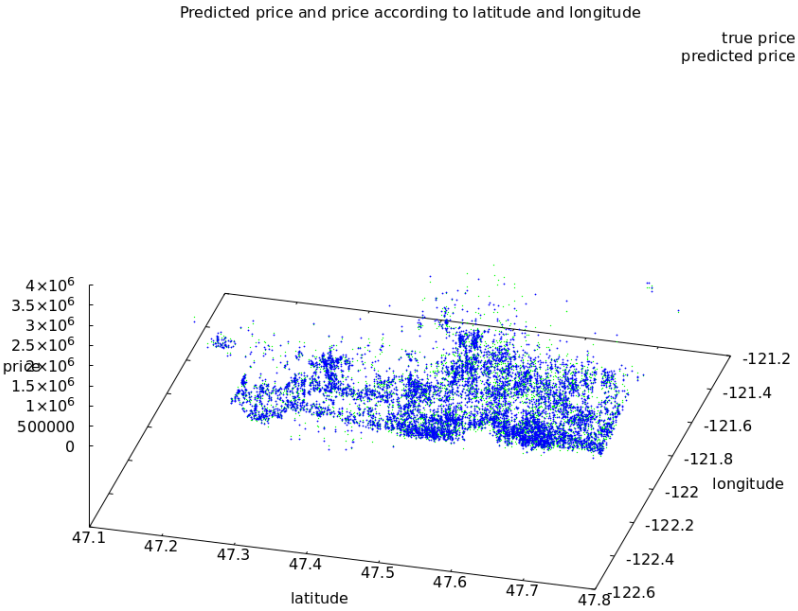

Figure 25: Latitude and longitude together. North is bottom-right. We can better see the most expensive neighbors. in the North-West region.

3. State of the framework and possible improvements

The framework has many useful features for basic Neural Networks and data preprocessing at that point, with a decent architecture and a simple API (as simple as Rust can go in my opinion).

Additionnaly to everything mentioned throughout this report, it can switch at compile-time between 32 bits and 64 bits floating point numbers, and between many Vectors and Matrices backends which are all CPU-bound. Such as the very fast `nalgebra` and `faer-rs` Rust libraries, and also a fully unit-tested custom implementation with another backend being the same implementation sped-up by a factor ~ 2.5 using parallelism with the `rayon` crate.

The framework is open-sourced on Github with 57 stars at the time of writing thanks to its mention in the Rust community's *subreddit* and also in a community weekly newsletter. I got feedback from experienced ML and Rust developers helping me pave the way towards my future goal: implementing a Rust-native GPU-bound backend for Vectors and Matrices using compute shaders and the WebGPU technology, (or at least plugging an existing one if it ends up being too difficult).

After that, I would like next to explore CNNs and RNNs.

4. Conclusion: Why (re)implementing NNs in Rust?

Machine Learning (ML) workflows almost always imply dynamic languages and notebooks, such as Matlab, Julia, R and Python. These versatiles and highly dynamic tools allow fast iterations and ease of use, enabling fast cutting-edge research. But they present a compromise over stability, maintainability, and sometimes performance.

In order for a ML workflow to be used at scale, low-level languages and GPU code is often used, such as C, C++, CUDA or OpenCL. These languages are very performant and well established, can interop with high-level tools, but they lack good memory safety, ease of use, universal and/or widely accepted standards for codebase management, documentation, dependency management, and unified testing practices. They also lack easy to use high-level abstractions for concurency, architecture, data structures and functional programming, which are very useful for ML workflows at scale, and would help reduce boilerplate, and further improve code readability and maintainability.

Rust aims to fill this niche for a low-level language feeling high-level, with its “Zero-Cost abstractions” philosophy, rich ecosystem of libraries, unified codebase management practices, and focus on performance, concurrency and memory safety. Rust is a very young language, but it has already gained some traction in the ML community, and even more in the graphics programming community, which gave birth to fast libraries for linear algebra, Cuda interop, and even Rust as a first-class language for GPU programming.

Creating performant but still high-level looking APIs in Rust, is absolutely possible and way easier than in C or C++ in my opinion. But still not trivial. It requires Rust-specific architecture skills, as Rust does not ressemble classic Object Oriented languages, and as usual planning, good library design, and a lot of back and forth API changes. This is why I initially struggled to find a good existing library or framework for my ML project in Rust. The existing ones compromising ease of use over extreme rigour and expressiveness, resulting in hairy APIs, that are hard to hack, or just grasp as a ML beginner like I am.

I wanted to create a NNs library that enabled me to create ML worflows looking high-level and still allowing low-level tweaking and average performance on the CPU on the long run. Something that Rust trully enabled me to achieve.

“What I cannot create, I do not understand.
Know how to solve every problem that has been solved.”

Richard Feynman

5. Appendix A: Code snippets

Here are some code snippets that you might find useful. They are excerpts from the framework's `0.1.1` version.

5. a. Specifying the model as code

```
// Including all features from some CSV dataset
let mut dataset_spec = Dataset::from_csv("kc_house_data.csv");
dataset_spec
    // Removing useless features for both the model & derived features
    .remove_features(&["id", "zipcode", "sqft_living15", "sqft_lot15"])
    // Setting up the price as the "output" predicted feature
    .add_opt_to("price", Out)
    // Setting up the date format
    .add_opt_to("date", DateFormat("%Y%m%dT%H%M%S"))
    // Converting the date to a date_timestamp feature
    .add_opt_to("date", AddExtractedTimestamp)
    // Excluding the date from the model
    .add_opt_to("date", Not(&UsedInModel))
    // Mapping yr_renovated to yr_built if = to 0
    .add_opt_to(
        "yr_renovated",
        Mapped(
            MapSelector::Equal(0.0.into()),
            MapOp::ReplaceWith(MapValue::Feature("yr_built".to_string())),
        ),
    )
    // Converting relevant features to their log10
    .add_opt(Log10.only(&["sqft_living", "sqft_above", "price"]))
    // Adding ^2 features of all input features
    // (including the added ones like the timestamp)
    .add_opt(AddSquared.except(&["price", "date"]).incl_added_features())
    // Filtering rows according to feature's outliers
    .add_opt(FilterOutliers.except(&["date"]).incl_added_features())
    // Normalizing everything
    .add_opt(Normalized.except(&["date"]).incl_added_features());

// Creating our layers
let h_size = dataset_spec.in_features_names().len() + 1;
let nh = 8;

let mut layers = vec![];
for i in 0..nh {
    layers.push(LayerSpec::from_options(&[
        OutSize(h_size),
        Activation(ReLU),
        Optimizer(adam()),
    ]));
}
let final_layer = LayerSpec::from_options(&[
    OutSize(1),
    Activation(Linear),
```

```
    Optimizer(adam()),
]);

// Putting it all together
let model = Model::from_options(&[
    Dataset(dataset_spec),
    HiddenLayers(layers.as_slice()),
    FinalLayer(final_layer),
    BatchSize(128),
    Trainer(Trainers::KFolds(8)),
    Epochs(300),
]);

// Saving it all
model.to_json_file("my_model_spec.json");
```

5. b. Preprocessing the data and training the model

```
// Loading a model specification from JSON
let mut model = Model::from_json_file("my_model_spec.json");

// Applying a data pipeline on it according to its dataset specification
let mut pipeline = Pipeline::basic_single_pass();
let (updated_dataset_spec, data) = pipeline
    .add(AttachIds::new("id"))
    .run("./dataset", &model.dataset);

let model = model.with_new_dataset(updated_dataset_spec);

// Training it using k-fold cross validation
// + extracting test & training metrics per folds & per epochs
// + extracting all predictions made during final epoch
let kfold = model.trainer.maybe_kfold().expect("We only do k-folds here!");
let (validation_preds, model_eval) = kfold
    .attach_real_time_reporter(|report| println!("Perf report: {:#?}", report))
    .run(&model, &data);

// Reverting the pipeline on the predictions & data to get interpretable values
let validation_preds = pipeline.revert_columnswise(&validation_preds);
let data = pipeline.revert_columnswise(&data);

// Joining the data and the predictions together
let data_and_preds = data.inner_join(&validation_preds, "id", "id", Some("pred"));

// Saving it all to disk
data_and_preds.to_file("my_model_preds.csv");
model_eval.to_json_file("my_model_evals.json");
```

5. c. K-folds detailed code

```
impl Trainer for KFolds {
    /// Runs the k-fold cross validation
    ///
    /// Assumes the data has all the columns corresponding to the model's dataset.
    ///
    /// Assumes both the data and the model's dataset include an id feature.
```

```
    fn run(self, model: &Model, data: &DataTable) -> (DataTable, ModelEvaluation) {
        // Running each fold in parallel requires Rust's compiler to satisfy
        // rigorous requirements
        // Which is why we create Mutexes (shared mutability accross threads) and
        // Arcs (shared variable lifetime garantees accross threads).
        let validation_preds = Arc::new(Mutex::new(DataTable::new_empty()));
        let model_eval = Arc::new(Mutex::new(ModelEvaluation::new_empty()));
        let reporter = Arc::new(Mutex::new(self.real_time_reporter));
        let mut handles = Vec::new();

        for i in 0..self.k {
            // Cloning the shared pointers before moving them
            // inside the thread closure to satisfy compiler's requirements
            let i = i.clone();
            let model = model.clone();
            let data = data.clone();
            let validation_preds = validation_preds.clone();
            let model_eval = model_eval.clone();
            let reporter = reporter.clone();

            let handle = thread::spawn(move || {
                let out_features = model.dataset.out_features_names();
                let id_column = model.dataset.get_id_column().unwrap();
                let mut network = model.to_network();

                let (train_table, validation) = data.split_k_folds(self.k, i);

                let (validation_x_table, validation_y_table) =
                    validation.random_order_in_out(&out_features);

                let validation_x =
validation_x_table.drop_column(id_column).to_vectors();
                let validation_y = validation_y_table.to_vectors();

                let mut fold_eval = FoldEvaluation::new_empty();
                let epochs = model.epochs;
                for e in 0..epochs {
                    let train_loss = model.train_epoch(e, &mut network, &train_table,
id_column);

                    let loss_fn = model.loss.to_loss();
                    let (preds, loss_avg, loss_std) =
                        network.predict_evaluate_many(&validation_x, &validation_y,
&loss_fn);

                    if let Some(reporter) = reporter.lock().unwrap().as_mut() {
                        reporter(KFoldsReport {
                            fold: i,
                            epoch: e,
                            train_loss,
                            validation_loss: loss_avg
                        });
                    }

                    let eval = EpochEvaluation::new(train_loss, loss_avg, loss_std);
```

```
                    // Save predictions from the final epoch
                    if e == model.epochs - 1 {
                        let mut vp = validation_preds.lock().unwrap();
                        *vp = vp.apppend(
                            &DataTable::from_vectors(&out_features, &preds)
                                .add_column_from(&validation_x_table, id_column),
                        )
                    };

                    fold_eval.add_epoch(eval);
                }
                model_eval.lock().unwrap().add_fold(fold_eval);
            });

            handles.push(handle);
        }

        for handle in handles.into_iter() {
            handle.join().unwrap();
        }

        let validation_preds = { validation_preds.lock().unwrap().clone() };
        let model_eval = { model_eval.lock().unwrap().clone() };

        (validation_preds, model_eval)
    }
}
```

5. d. SGD, Momentum, Adam implementations

```
impl SGD {

    // [...]

    pub fn update_parameters(&mut self, epoch: usize, parameters: &Matrix,
parameters_gradient: &Matrix) -> Matrix {
        let lr = self.learning_rate.get_learning_rate(epoch);
        parameters.component_sub(&parameters_gradient.scalar_mul(lr))
    }
}
impl Momentum {

    // [...]

    pub fn update_parameters(&mut self, epoch: usize, parameters: &Matrix,
parameters_gradient: &Matrix) -> Matrix {
        let lr = self.learning_rate.get_learning_rate(epoch);

        let v = if let Some(v) = self.v.clone() {
            v
        } else {
            let (nrow, ncol) = parameters_gradient.dim();
            Matrix::zeros(nrow, ncol)
        };
```

```
        let v =
v.scalar_mul(self.momentum).component_add(&parameters_gradient.scalar_mul(lr));

        let new_params = parameters.component_sub(&v);
        self.v = Some(v);
        new_params
    }
}
impl Adam {

    // [...]

    pub fn update_parameters(
        &mut self,
        epoch: usize,
        parameters: &Matrix,
        parameters_gradient: &Matrix,
    ) -> Matrix {
        let alpha = self.learning_rate.get_learning_rate(epoch);

        let (nrow, ncol) = parameters_gradient.dim();

        if self.m.is_none() {
            self.m = Some(Matrix::zeros(nrow, ncol));
        }
        if self.v.is_none() {
            self.v = Some(Matrix::zeros(nrow, ncol));
        }
        let mut m = self.m.clone().unwrap();
        let mut v = self.v.clone().unwrap();

        let g = parameters_gradient;
        let g2 = parameters_gradient.component_mul(&parameters_gradient);

        m = (m.scalar_mul(self.beta1)).component_add(&g.scalar_mul(1.0 -
self.beta1));
        v = (v.scalar_mul(self.beta2)).component_add(&g2.scalar_mul(1.0 -
self.beta2));

        let m_bias_corrected = m.scalar_div(1.0 - self.beta1);
        let v_bias_corrected = v.scalar_div(1.0 - self.beta2);

        let v_bias_corrected = v_bias_corrected.map(Scalar::sqrt);

        parameters.component_sub(
            &(m_bias_corrected.scalar_mul(alpha))
                .component_div(&v_bias_corrected.scalar_add(self.epsilon)),
        )
    }
}
```

5. e. Dense layer and backpropagation implementation

```
impl Layer for DenseLayer {
    /// `input` has shape `(i, n)` where `i` is the number of inputs and `n` is the
number of samples.
```

```rust
    ///
    /// Returns output which has shape `(j, n)` where `j` is the number of outputs
and `n` is the number of samples.
    fn forward(&mut self, input: Matrix) -> Matrix {
        // Y = W . X + B
        let mut res = self.weights.dot(&input);

        let biases = self.biases.get_column(0);

        // Adding the i^th bias to the i^th row on all columns
        res.map_indexed_mut(|i, _, v| v + biases[i]);

        self.input = Some(input);
        res
    }

    /// `output_gradient` has shape `(j, n)` where `j` is the number of outputs and
`n` is the number of samples.
    ///
    /// Returns `input_gradient` which has shape `(i, n)` where `i` is the number of
inputs and `n` is the number of samples.
    fn backward(&mut self, epoch: usize, output_gradient: Matrix) -> Matrix {
        let input = self.input.clone().unwrap();

        let weights_gradient = &output_gradient.dot(&input.transpose());

        let biases_gradient =
Matrix::from_column_vector(&output_gradient.columns_sum());

        let input_gradient = self.weights.transpose().dot(&output_gradient);

        self.weights =
            self.weights_optimizer
                .update_parameters(epoch, &self.weights, &weights_gradient);
        self.biases =
            self.biases_optimizer
                .update_parameters(epoch, &self.biases, &biases_gradient);

        input_gradient
    }
}
```

6. Appendix B: Codebase overview

6. a. Lines of Code

Language	Files	Lines	Blank	Comment	Code
JSON	125	28866	0	0	28866
Rust	58	5855	848	386	4621
Markdown	4	258	49	0	209
Toml	4	86	10	5	71
Total	191	35065	907	391	33767

6. b. Project structure

```
├── activation
│   ├── hbt.rs
│   ├── linear.rs
│   ├── mod.rs
│   ├── relu.rs
│   ├── sigmoid.rs
│   └── tanh.rs
├── benchmarking.rs
├── dataset.rs
├── datatable.rs
├── initializers.rs
├── layer
│   ├── dense_layer.rs
│   ├── full_layer.rs
│   └── mod.rs
├── learning_rate
│   ├── inverse_time_decay.rs
│   ├── mod.rs
│   └── piecewise_constant.rs
├── lib.rs
├── linalg
│   ├── faer_matrix.rs
│   ├── matrix.rs
│   ├── mod.rs
│   ├── nalgebra_matrix.rs
│   └── rayon_matrix.rs
├── loss
│   ├── mod.rs
│   └── mse.rs
├── main.rs
├── model.rs
├── network.rs
├── optimizer
│   ├── adam.rs
│   ├── mod.rs
│   ├── momentum.rs
```

```
│   └── sgd.rs
├── pipelines
│   ├── attach_ids.rs
│   ├── extract_months.rs
│   ├── extract_timestamps.rs
│   ├── feature_cached.rs
│   ├── filter_outliers.rs
│   ├── log_scale.rs
│   ├── map.rs
│   ├── mod.rs
│   ├── normalize.rs
│   └── square.rs
├── trainers
│   ├── kfolds.rs
│   └── mod.rs
└── vec_utils.rs
```