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 performant NN model trained with supervised learning on the King County House dataset [1]. At first I struggled with the limited features of my framework and my limited intuition with NNs based approaches. Trying around 100 small models with varying success, getting stuck trying irrelevant 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. Then, using my framework configured with almost the same defaults as the Keras Python library, I finally recreated an existing performant Python Keras workflow [2] for King County houses price inference, and then tweaked the author's model, obtaining a better R^2 at around 0.9. Finally, 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 some 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 and a link to the source code in the appendix.

Contents

1. Introducing NNs: Learning the xor function

Learning the xor function is a basic first step when implementing NNs [3]. 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.

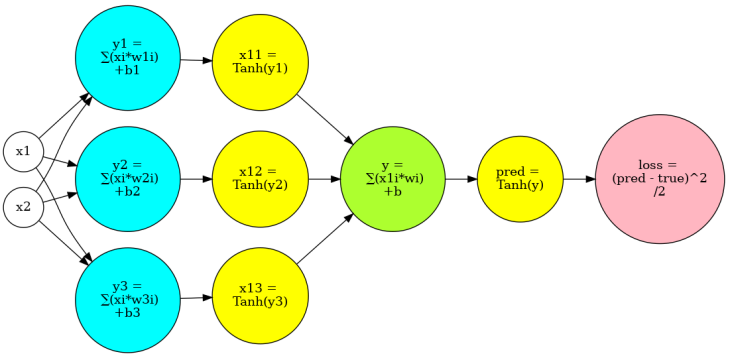

Figure 1: Network used to learn the xor function.

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 [3].

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

1. a. a. Explaining the forward pass

Imagine solving a regression problem for a dataset composed of many observations. Each observation comprises i values (in the case of xor the observations would be the four different cases in its truth table, and the values a and b), and we want to predict j values (in the case of xor one value: $a \ \mathrm{xor} \ b$, but we need to generalize to problems with more than one output, such as classification problems that use *one-hot encoding*).

We can first look at it like a black box: The network's first layer is fed with an observation's i values, or *features* $\underset{i^{(0)}\times 1}{X}^{(0)} = \begin{pmatrix} x_0 \\ x_1 \\ \cdots \\ x_{i^{(0)}} \end{pmatrix}$, and the network's last layer produces j predictions values $\underset{j\times 1}{\hat{Y}} = \begin{pmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \cdots \\ \hat{y}_j \end{pmatrix}$.

What happens under the hood is that the network executes a *forward pass*, which for each layer n from first 0 to last $(N-1)$, computes an intermediary output matrix $\underset{j^{(n)}\times 1}{Y}^{(n)}$ by doing the sum of its weights $\underset{j^{(n)}\times i^{(n)}}{W}^{(n)}$ matrix-multiplied by its intermediary input matrix $\underset{i^{(n)}\times 1}{X}^{(n)}$ plus its biases $\underset{j^{(n)}\times 1}{B}^{(n)}$ [4].

$$\underset{j^{(n)}\times 1}{Y}^{(n)} = \underset{j^{(n)}\times i^{(n)}}{W}^{(n)} . \underset{i^{(n)}\times 1}{X}^{(n)} + \underset{j^{(n)}\times 1}{B}^{(n)}$$

But since the input of the n^{th} layer is the output of the $(n-1)^{\text{th}}$ layer, the input size $i^{(n)}$ of the layer is the output size $j^{(n-1)}$ of the previous layer, and we can define that whole forward pass in a recurrent manner and remove X and j in the process:

$$\underset{i^{(n+1)}\times 1}{Y}^{(n)} = \underset{i^{(n+1)}\times i^{(n)}}{W}^{(n)} . \underset{i^{(n)}\times 1}{Y}^{(n-1)} + \underset{i^{(n+1)}\times 1}{B}^{(n)}$$

Obtaining after the last $(N-1)^{\text{th}}$ layer:

$$\underset{j\times 1}{Y}^{(N-1)} = \underset{j\times i^{(N-1)}}{W}^{(N-1)} . \underset{i^{(N-1)}\times 1}{Y}^{(N-2)} + \underset{i^{(N-1)}\times 1}{B}^{(N-1)}$$

Which is our prediction vector $\underset{j\times 1}{\hat{Y}}$ containing all our $\hat{y}_j$ predictions deduced from the $\underset{i\times 1}{X}$ observation and its x_i features.

This is fairly straight-forward to implement using a linked list or an array of some datastructure storing the weights and biases, and an overarching datastructure feeding the outputs to the inputs one after the others [3]. What is trickier and can lead to hard to debug runtime issues, is messing up matrices sizes and getting lost between matrices, vectors, column vectors, row-leading matrices, not knowing anymore what the rows and columns even are. You can get lost easily in all these different matrix implementation details. Which is why I write the sizes below the matrices in the formulas.

1. a. b. Intuition on why NNs work for complex regression problems

If we decompose the matrix operations and try to reason about individual y and x we see that the $i^{(n+1)}$ outputs $y^{(n)}$ of the n^{th} layer's $i^{(n+1)}$ neurons are given as a linear combination of the layer's $i^{(n)}$ inputs $x^{(n)}$ (which are the outputs $y^{(n-1)}$ of the previous layer), multiplied by its $i^{(n+1)} \times i^{(n)}$ weights $w^{(n)}$ plus its $i^{(n+1)}$ biases $b^{(n)}$. Which after substitution of the $x^{(n)}$ by the equivalent $y^{(n-1)}$ gives:

$$y_0^{(n)} = y_0^{(n-1)} . w_{0,0}^{(n)} + y_1^{(n-1)} . w_{1,0}^{(n)} + ... + y_{i^{(n)}}^{(n-1)} . w_{i^{(n)},0}^{(n)} + b_0^{(n)}$$

$$y_1^{(n)} = y_0^{(n-1)} . w_{0,1}^{(n)} + y_1^{(n-1)} . w_{0,1}^{(n)} + ... + y_{i^{(n)}}^{(n-1)} . w_{i^{(n)},0}^{(n)} + b_1^{(n)}$$

$$...$$

$$y_{i^{(n+1)}}^{(n)} = y_0^{(n-1)} . w_{0,i^{(n+1)}}^{(n)} + y_1^{(n-1)} . w_{0,i^{(n+1)}}^{(n)} + ... + y_{i^{(n)}}^{(n-1)} . w_{i^{(n)},i^{(n+1)}}^{(n)} + b_{i^{(n+1)}}^{(n)}$$

As we can see, a Neural Network is just a chain of regression models. Instead of using a simple regression model computing the output as a linear combination of weighted inputs (weighted because not all the inputs are correlated the same to the output), plus some bias (the ideal regression line may not cross $(0, 0)$ so we need some bias), we stack regression models one after the other to build progressively a more and more complex model. We can see them as intermediary patterns in the data that would be more useful than the initial inputs in order to guess the final output using some linear equation.

Intuitively, you know you can't guess the result of the xor function by just weighting the a and b inputs and that's all. You have to make a table of truth where you look at both a

and b together. You look at three intermediary patterns essentially: (1) whether a and b are both 1, (2) whether a and b are both 0, and (3) whether a and b are different. So from two inputs a and b you have to construct mentally 3 intermediary facts $(1), (2), (3)$ which you then combine logically to guess the final result. If $(1) = 1$ then the result is 0, if $(2) = 1$ then the result is also 0 and if $(3) = 1$ then the result is 1.

The Network just weights three internal facts computed by the hidden layer. More complex problems may require more intermediary patterns and may require to stack more layers of successive pattern extractions to find the answer. Training the model is just tweaking the biases and weights that react to these patterns in order to maximize our network's ability to guess the result.

But as we've seen, the output has a linear relationship to the input, which may not represent how the reality works. So once we have the $y^{(n)}$ we pass it through a non-linear *activation function*, $\tanh$ in our case, which will make the model non-linear. Many activation functions exist and have different properties, relevant to certain models.

1. a. c. Backward pass

We wish to minimize the prediction error, both with the current observations and the future observations, and this without *overfitting* to our current observations. So we compute the *loss* of the activated prediction relative to the true value, using for example a *Mean Squared Errors (MSE) loss function*, which is a common loss function [4] helping to converge towards both a low variance and bias model. Simpler formulas for MSE exist, but the one that describes the implementation details well is [3]:

$$\mathrm{MSE}\Big(\underset{J\times 1}{Y}, \underset{J\times 1}{\hat{Y}}\Big) = \frac{1}{J}\sum_{j=0}^{J-1}\Big(y_j - \hat{y}_j\Big)^2$$

And for N predictions $\underset{N\times J\times 1}{P} = \left(\underset{J\times 1}{\hat{Y}_0} \cdots \underset{J\times 1}{\hat{Y}_{N-1}}\right)$ and associated true values $\underset{N\times J\times 1}{T} = \left(\underset{J\times 1}{Y_0} \cdots \underset{J\times 1}{Y_{N-1}}\right)$ in order to know how the model performs over the whole training and validation dataset:

$$\mathrm{MSE}^N\Big(\underset{N\times J\times 1}{T}, \underset{N\times J\times 1}{P}\Big) = \frac{1}{N}\sum_{n=0}^{N-1}\mathrm{MSE}\left(\underset{J\times 1}{T_n}, \underset{J\times 1}{P_n}\right)$$

We want to minimize the loss function, hence find weights and biases that react to appropriate patterns in the different layers of our network in such a way that it outputs a result resulting in a low MSE.

In order to progress towards that goal, we execute a *backward pass* using the *Stochastic Gradient Descent (SGD)* algorithm [4]. This optimization algorithm will make the model converge towards weights and biases that minimize the loss function.

SGD 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) $\frac{\delta E}{\delta W}$ and $\frac{\delta E}{\delta B}$, and updates the parameters in the opposite direction of their gradient, multiplied by a *learning rate* r hyperparameter. As an example with the biases: $B = B - r \times \frac{\delta E}{\delta B}$. It then passes the gradient of the loss with respect to its inputs (e.g. the previous layer's output) $\frac{\delta E}{\delta X}$, so that the previous layers can repeat the same process for themselves. This algorithm is called *backpropagation* [4].

If we consider that $X^{(n+1)} = Y^{(n)} = L^{(n)}\left(X^{(n)}\right) = L^{(n)}\left(Y^{(n-1)}\right)$ whe can write:

$$E\left(X^{(0)}\right) = \text{MSE}\left(L^{(N-1)}\left(...L^{(1)}\left(L^{(0)}\left(X^{(0)}\right)\right)\right)\right))$$

Or:

$$E\left(X^{(0)}\right) = \text{MSE} \circ L^{(N-1)} \circ ... \circ L^{(1)} \circ L^{(0)}$$

And:

$$L^{(n)} = L^{(n-1)} \circ ... \circ L^{(1)} \circ L^{(0)}$$

Hence:

$$E\left(X^{(n)}\right) = \text{MSE} \circ L^{(N-1)} \circ ... \circ L^{(n)}$$

Using the chain rule to compute $(f \circ g)'$ whe can compute the gradient of the error E with respect to any layer's input $X^{(n)}$, therefore output $Y^{(n-1)}$, but also weights and biases since they intervened in computing $Y^{(n-1)}$, and apply SGD on the learnable parameters. We can also put our activation like any other layer $L^{(n)}$ in the formulas, and therefore use its derivative when computing the gradients.

This can be implemented as the mirror of the forward pass' implementation. A structure feeds $\frac{\delta E}{\delta X^{(n)}}$, to the previous $n-1$ layer, which becomes its $\frac{\delta E}{\delta Y}$, from which it can compute the other gradients using these formulas which can be proved using the chain rule [3]:

$$\underset{j \times i}{\frac{\delta E}{\delta W}} = \underset{j \times 1}{\frac{\delta E}{\delta Y}} \cdot \underset{1 \times i}{\left(\underset{i \times 1}{X}\right)^t}$$

$$\underset{j \times 1}{\frac{\delta E}{\delta B}} = \underset{j \times 1}{\frac{\delta E}{\delta Y}}$$

$$\underset{i \times 1}{\frac{\delta E}{\delta X}} = \underset{i \times j}{\left(\underset{j \times i}{W}\right)^t} \cdot \underset{j \times 1}{\frac{\delta E}{\delta Y}}$$

We repeat both the forward and the backward passes for e *epochs*. After the last epoch, hopefully, the loss converged to a local minimum and predictions start looking quite good.

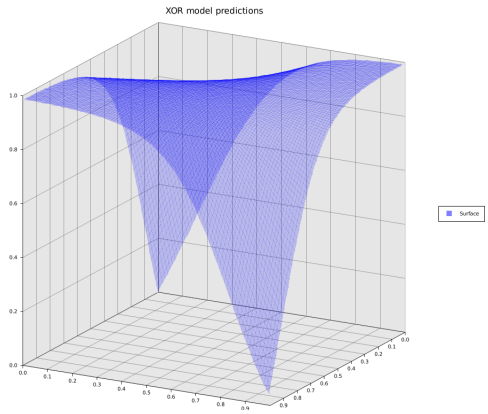

Figure 2: Predictions for inputs ranging from $(0.0, 0.0)$ to $(1., 1.)$

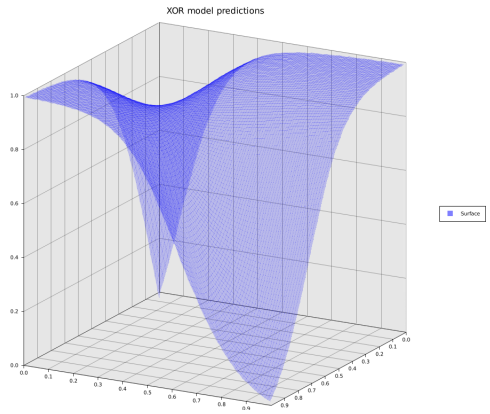

Figure 3: Not all trainings lead to the same predictions. This is another local minima found, hence the new shape

As we can see, the NN which had weights and biases initialized at random following $\mathcal{U}_{[-1,1]}$, tried to generalize based on the limited observations it received, and therefore can predict what the *non-existent* value of xor would be for any input tuple in the $[0, 1]^2$ interval [3]. But since the starting weights and biases are set at random, it converges to different minima each time, which leads to different predictions.

1. b. Adding Learning Rate Decay and Dropouts

I then added *learning rate decay* [4, 5] 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 [6]:

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

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 [4]. Finding the right value is tricky, as too high values can lead to the model converging too slowly.

I also added *dropouts* [7, 8] which is a *regularization* [9] technique hence used to prevent overfitting by randomly dropping nodes of each layer with a probability $1 - p$ during the forward pass (by setting their input or activation to 0), and adapting the backward pass accordingly.

In order to apply dropouts during the forward pass for a layer with inputs $\underset{I\times 1}{X}$ we generate a mask $\underset{I\times 1}{M}$:

$$\underset{I\times 1}{M} = \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{I-1} \end{pmatrix} \qquad \text{with } \forall i \in \{0, 1, ..., (I-1)\} : d_i \sim \text{Bernouilli}(p)$$

Then we compute the partially dropped input matrix using element-wise product:

$$\underset{I\times 1}{\widetilde{X}} = \underset{I\times 1}{X} \odot \underset{I\times 1}{M}$$

To compute the predictions during the test phase, we attenuate the weights by temporally doing $W_{\text{test}} = pW$.

After the backward pass on the layer using Dropouts, we compute a correction of $\frac{\delta E}{\delta X}$:

$$\underset{I\times 1}{\frac{\delta E}{\delta \widetilde{X}}} = \underset{I\times 1}{\frac{\delta E}{\delta X}} \odot \underset{I\times 1}{M}$$

Before sending it to the previous layer for it to use it as its $\frac{\delta E}{\delta Y}$.

In a way it simulates learning from a different, simpler model at each epoch, building more varied and robust features [7]. 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:

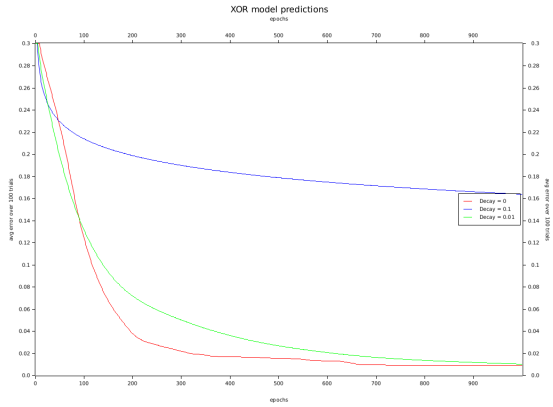

Figure 4: High decay (blue) learns too slowly

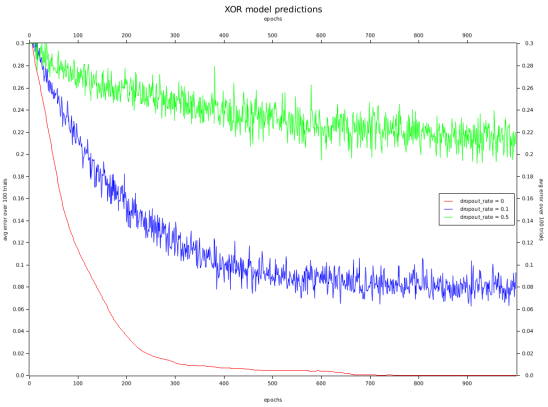

Figure 5: Dropout (blue & green) make simple models worse by dropping too much information

I slowly started realizing how over-engineering by sticking fancy features together wouldn't help.

2. Price inference on the King County Houses dataset

During this next segment I will describe how I gradually updated my framework in order to solve the King County House price inference problem [1], with a final score of $R^2 > 0.9$.

2. a. Normalization and trying initial hyperparameters

With 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 [10].

So I got started with *feature scaling* [10] by normalizing the data features using *min-max normalization* [11]: 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.

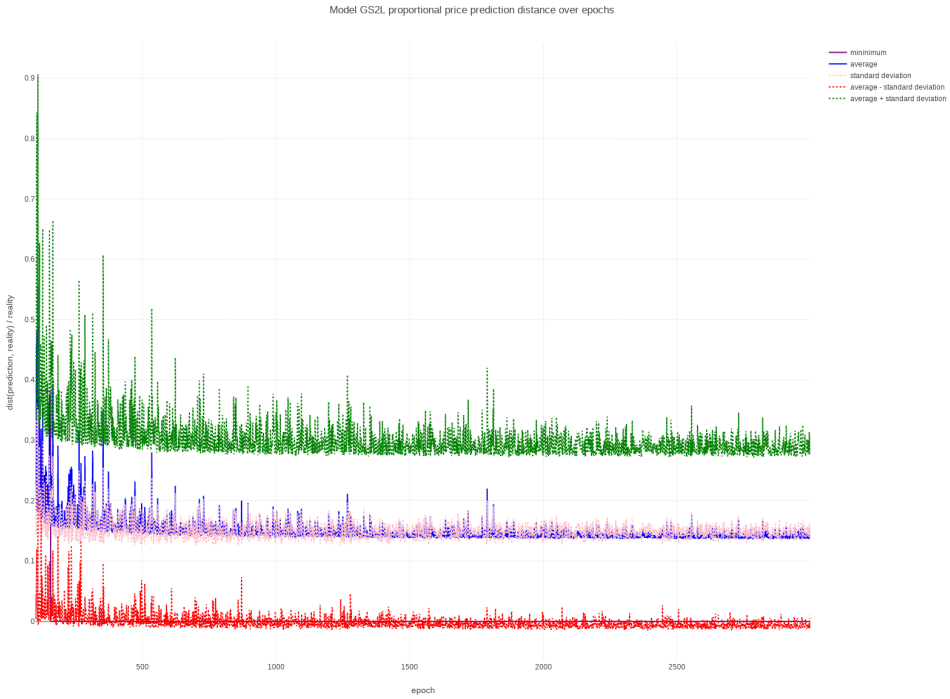

Figure 6: The $\pm 20\%$ average proportional distance, with $\pm 20\%$ standard deviation is not great. Is this better than random guessing?

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 [12], 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)

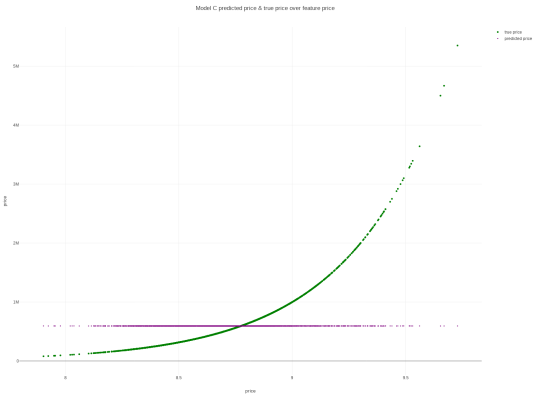

Figure 8: This model learned it even better, it now always predicts the exact same value that statistically makes sense.

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 believed at that point 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. Which I ended up demonstrating at the end of the project. Although they seemed 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, pre-process 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 foots*, using a small model consisting of one 9 nodes hidden layer, using SGD with a constant learning rate I started getting better results:

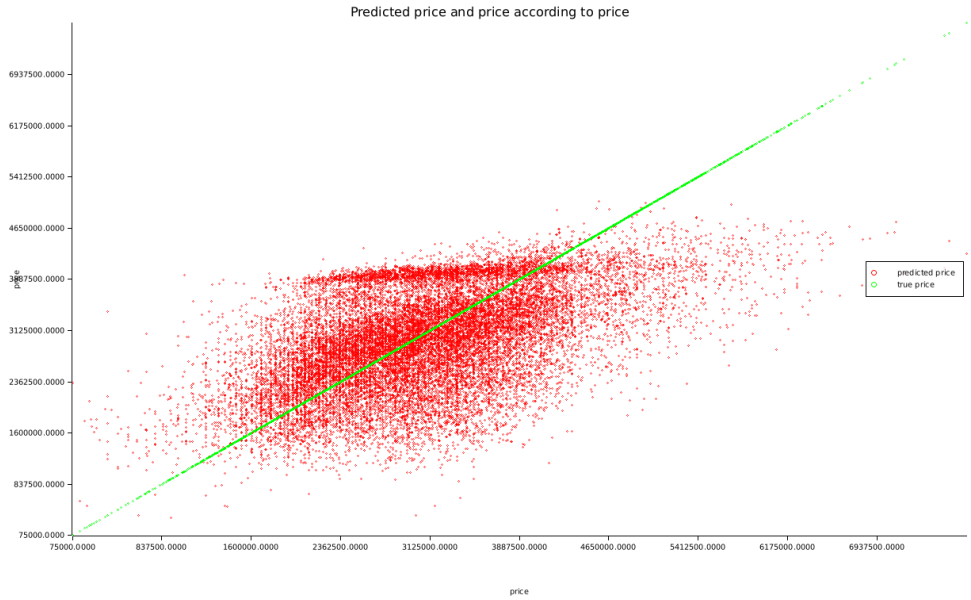

Figure 9: The predicted prices follow the true prices much better.

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

I added the possibility to train with *mini-batches* [13, 14, 15]. 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.

Therefore with mini-batches of size S I now had this forward pass formula (notice how the sizes changed):

$$\underset{i^{(n+1)}\times S}{Y}^{(n)} = \underset{i^{(n+1)}\times i^{(n)}}{W}^{(n)} . \underset{i^{(n)}\times S}{Y}^{(n-1)} + \underset{i^{(n+1)}\times 1}{B}^{(n)} . \underset{1\times S}{(1\ ...)}$$

This MSE formula (omitting the derivative which changed in a similar manner):

$$\text{MSE}\left(\underset{J\times S}{Y}, \underset{J\times S}{\hat{Y}}\right) = \frac{1}{J\times S}\sum_{s=0}^{S-1}\sum_{j=0}^{J-1}\left(y_{j,s} - \hat{y}_{j,s}\right)^2$$

And these backward pass formulas:

$$\underset{j\times i}{\frac{\delta E}{\delta W}} = \underset{j\times S}{\frac{\delta E}{\delta Y}} . \underset{S\times i}{\left(\underset{i\times S}{X}\right)^t}$$

$$\underset{j\times 1}{\frac{\delta E}{\delta B}} = \underset{j\times S}{\frac{\delta E}{\delta Y}} . \underset{S\times 1}{\begin{pmatrix}1\\ \vdots \\ 1\end{pmatrix}}$$

$$\underset{i\times S}{\frac{\delta E}{\delta X}} = \underset{i\times j}{\left(\underset{j\times i}{W}\right)^t} . \underset{j\times S}{\frac{\delta E}{\delta Y}}$$

Activations are easier since they just map every cell of a matrix, nothing had to change.

This was a difficult refactor as it had repercussions on almost the whole code. It made the math seen in the previous sections trickier and the matrices sizes more error-prone. But it indeed made the training significantly faster.

I also realized during my experiments that the higher the batch size, the more I had to decrease the learning rate. I haven't found literature on that yet. My intuition is that since we now used the sum of the samples gradients during backpropagation, it made the extreme gradients more extreme.

2. f. Adding Momentum SGD

Then, I added the possibility to train with *momentum SGD* [15]. It changes how we update learnable parameters. Instead of subtracting the gradient, we keep a learning rate r and its decay parameters if decay is enabled, and we introduce an hyperparameter v that weights how much we also consider the previous gradient in the next update of our parameters. Let's show the formulas for updating the weights, but it is similar for the biases:

So first at epoch $e = 0$ we define an initial momentum M:

$$M^{(0)} = \left(\frac{\delta E}{\delta W}\right)^{(e-1)}_{j \times i} = \begin{pmatrix} 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix}$$

And then for each epoch e we do:

$$M^{(e)} = vM^{(e-1)} + r\left(\frac{\delta E}{\delta W}\right)^{(e)}$$

$$W^{(e)} = W^{(e-1)} - M^{(e)}$$

Like a ball rolling down a hill, the momentum helps the model get out of local minima and reach a better local or global minimum.

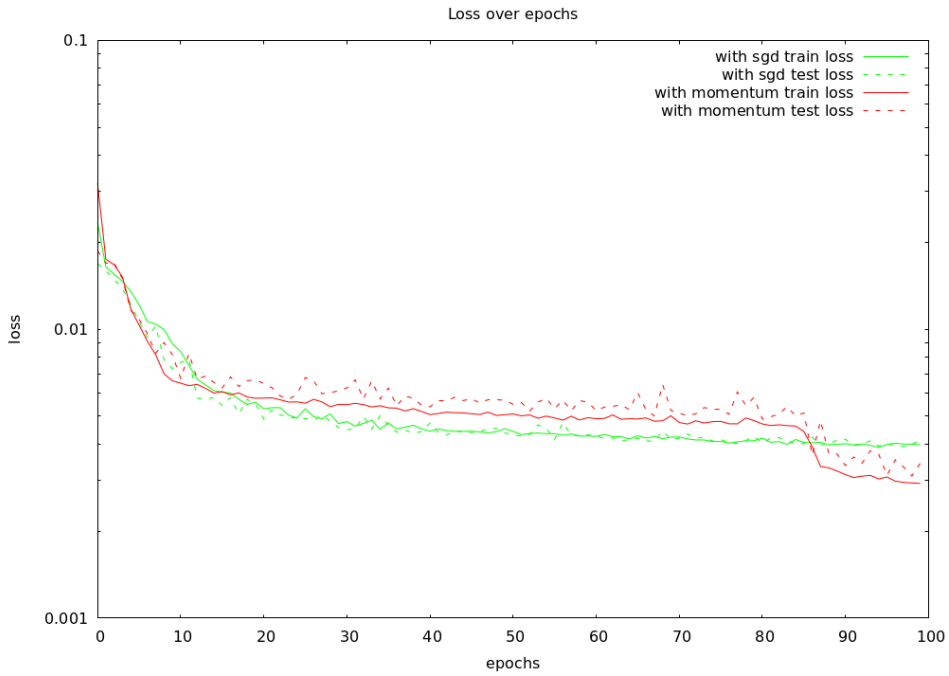

Figure 11: SGD vs Momentum on the final model. Momentum ends up finding a better minimum. Since epoch ~ 25 SGD floored at 0.004 training loss. Momentum had all its folds at 0.002 but had one badly performing fold at 0.016 which made it all worse on average. After 75 epochs thought this bad fold got unstuck.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

2. g. ReLU, squared features, Glorot and reaching stability

I wanted to make the switch to ReLU activation [16] because it had a reputation of converging faster [17]. But I was struggling with it because it was giving very unstable results with my current models at that time.

But I discovered I was initializing the biases with a random value between -1 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 the gradient helping the bias enough to reach a positive value. In other words, it made the model way too sparse to be able to densify itself enough ever again. In opposition to $\tanh$ which is symmetric and therefore treats negative values as positive ones.

Also, ReLU is linear in both negative and positive inputs (but not linear as a whole by definition) so maybe it did not learn non-linear relationships well. I added squares of the input features to the input data at that point. Maybe the squared nature of the input helped it learn.

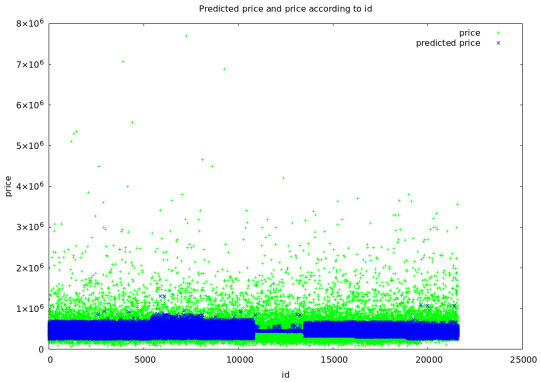

Figure 12: With ReLU with squared features and biases initialized at 0

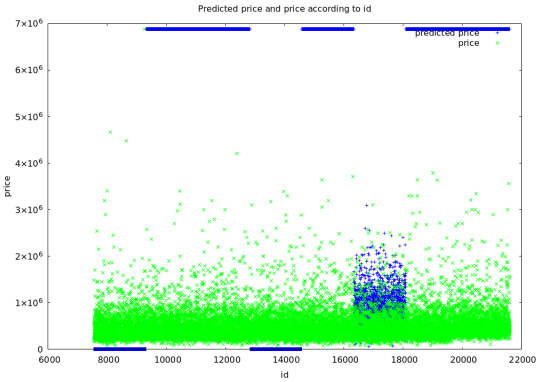

Figure 13: With ReLU with squared features and biases initialized in $[-1, 1]$

I also started implementing the *Glorot Uniform* weights initialization, since it was Keras' default weights initializer. I read the original paper [18], implemented it:

$$W^{(0)}_{j \times i} \sim \mathcal{U}^{j \times i}_{[-l,l]} \qquad \text{With } l = \sqrt{\frac{6}{j+i}}$$

But I struggled to really see why it works. It does improve the models' stability and performance thought, whether using ReLU or $\tanh$ activation.

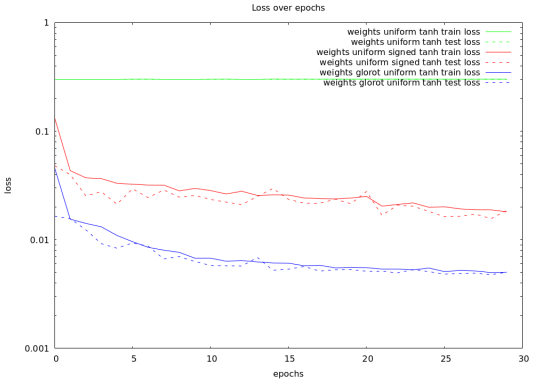

Figure 14: Comparing uniform, uniform signed and Glorot uniform weights initialization with $\tanh$. Glorot outperforms the other two.

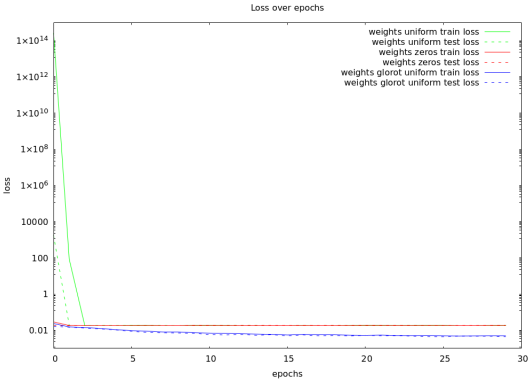

Figure 15: Comparing uniform, zeros and Glorot uniform weights initialization with ReLU. Glorot outperforms the other two.

The other great advantage of using ReLU is that it is a very fast activation function [17]. It made my network way faster than with $\tanh$. And Glorot is the best initialization I found when using ReLU. So now I had the best of both worlds.

2. h. Log scaling and outliers filtering

One issue I had with the data was its skewed distribution which did not help the model. A fix was to log-scale the features biased towards lower values, such as the *price* or the *squared foots* 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 [19], which removes a row if one of it features' is above or below 1.5 times that feature's *interquartile range (IQC)*.

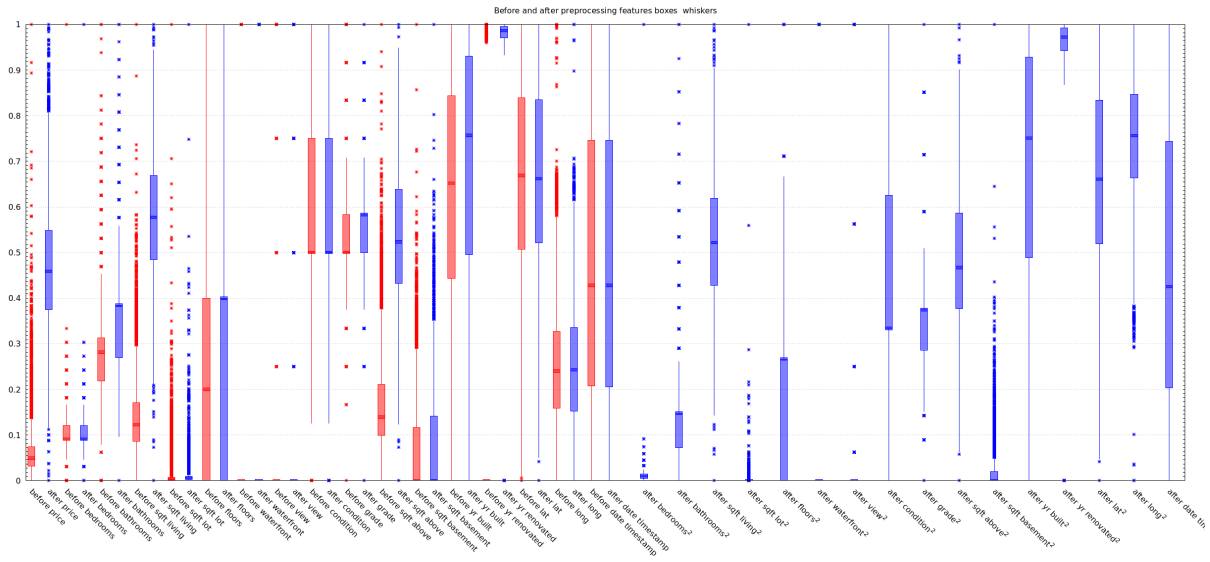

Figure 16: Box and whisker plots of the features' distributions before (red) and after(blue) the full preprocessing pipeline. The log scaled features are much more normally distributed. The outliers are also much less extreme.

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.

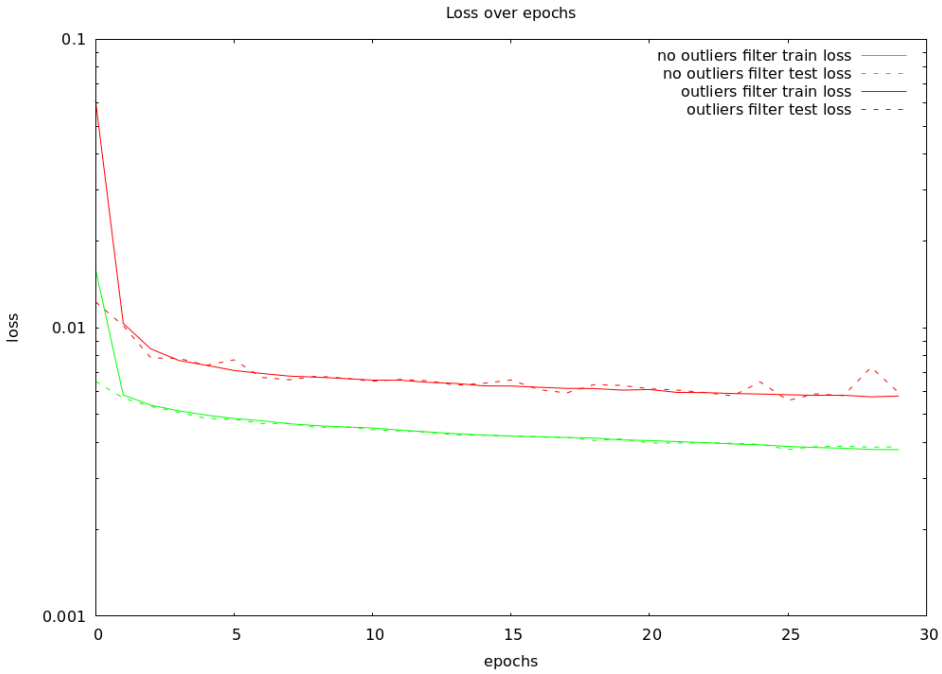

Figure 17: The red model was trained on ~ 750 rows, versus ~ 2625 for the red one, and only has an average of ~ 0.002 more loss.

2. i. Implementing Adam and comparison with 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 [20]. 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. After some hyperparameter fine-tuning maybe it would perform better, or maybe I should try optimized versions of it [21] but I haven't got time to try that yet.

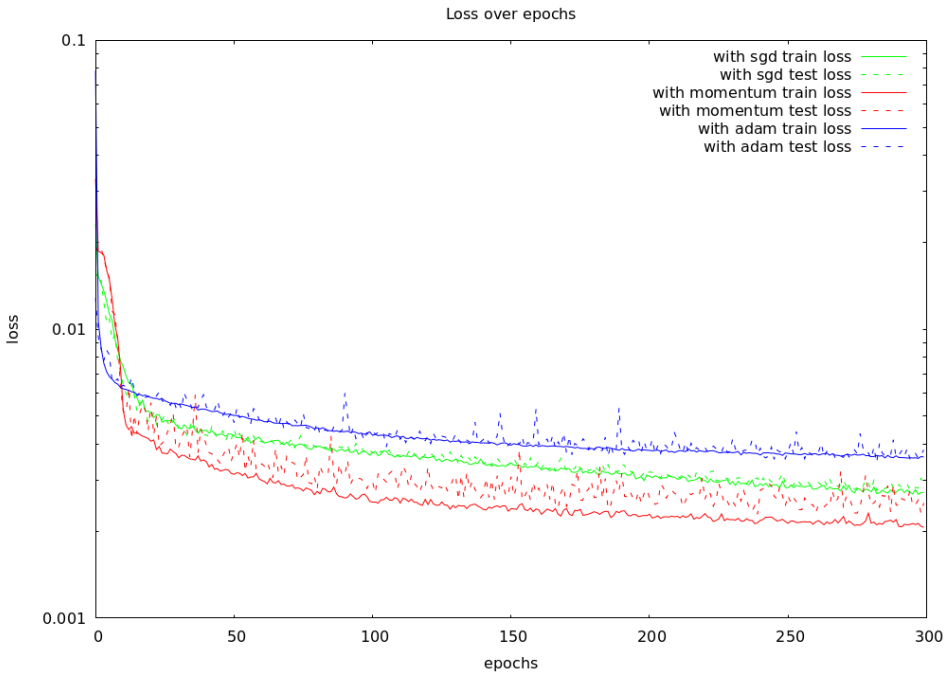

Figure 18: Adam, SGD and Momentum

2. j. Final results

2. j. a. Architecture and k-folds training

After all these tweaks and experiments, I ended up with a model consisting of 8 hidden layers of as many neurons as inputs $+1$ (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 disabled outliers filtering on the preprocessing pipeline to have better accuracy.

I trained it for 500 epochs on 8 folds with mini-batches of size 128.

Considering the framework is CPU-bound, it trained on my personal computer for 43 minutes when also computing various performance metrics for each epoch, and for 32

minutes without additional computations. The data preprocessing pipeline took an insignificant amount of time to run prior to the training.

Here you can see the average loss over the 8 folds throughout the training:

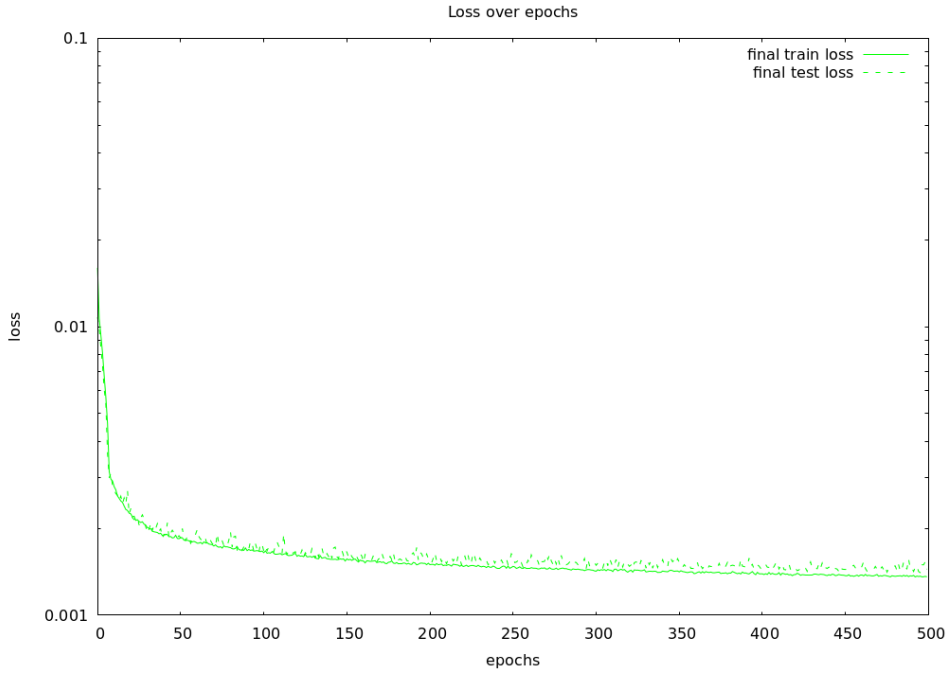

Figure 19: Loss over epochs

2. j. b. Computing R^2 and comparison with a similar Keras workflow

I 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 [22]:

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

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).

The best of the 8 folds models attains an $R^2 = 0.9$.

It is slightly better than the Python Keras workflow I got inspiration from [2] at $R^2 = 0.88$, which used a similar architecture, trained for 500 epochs as well with mini-batches of size 128.

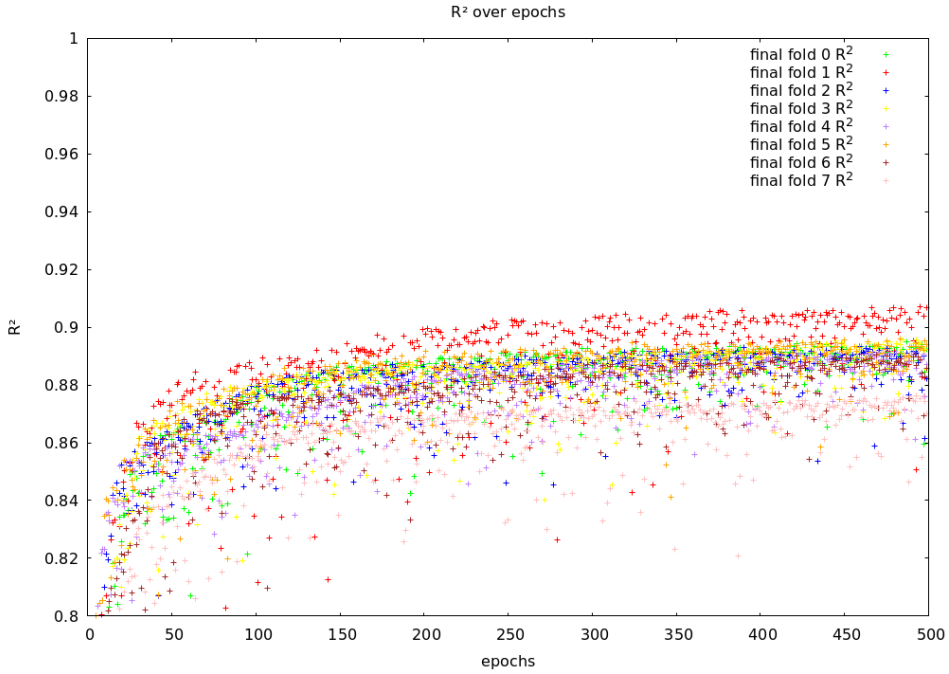

Figure 20: Folds R^2 over epochs

To the workflow's defense: it did not pick the best model out of a k-folds training like I did. It only did one $80\%/20\%$ train/test data split. For comparison's sake, my average R^2 over the 8 folds stands at 0.88.

Also, the workflow used Adam optimization (Keras' default) and outliers filtering, both I found to not benefit the model. And it used 35 features while I used 34. Its features included an engineered feature based on the house's grade and year built which I couldn't replicate, which the author found out had a 3% relative importance in predicting the price (I'll explain that metric later, but 3% is quite significant). But unlike me it didn't use the timestamp not the month of the sale, which ended up being relatively very unimportant ($< 0.5\%$).

Out of curiosity I also tried averaging my 8 models' weights and biases. It gave me a model predicting always the same statistically average price. I wonder whether merging the folds models can be done in a different way in order to extract the best performing weights and biases from all the models.

2. j. c. Visualizing the model's predictions

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

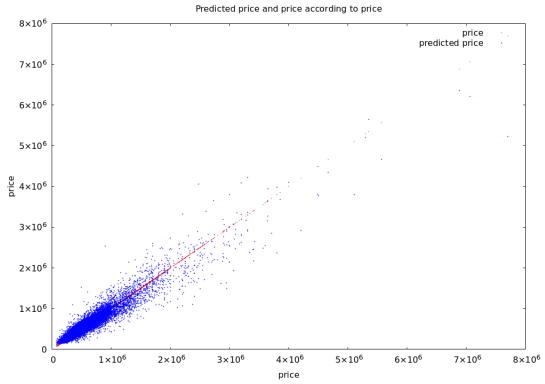

Figure 21: Predictions grow with actual value as expected. Higher prices seem to be harder to predict. Maybe they are too shallow, and the dataset lacks high price examples.

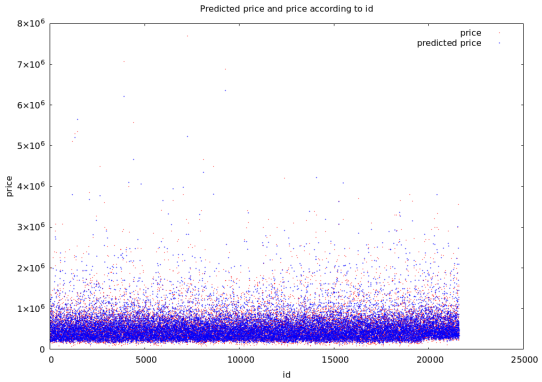

Figure 22: The unstable folds issue is far gone at that point

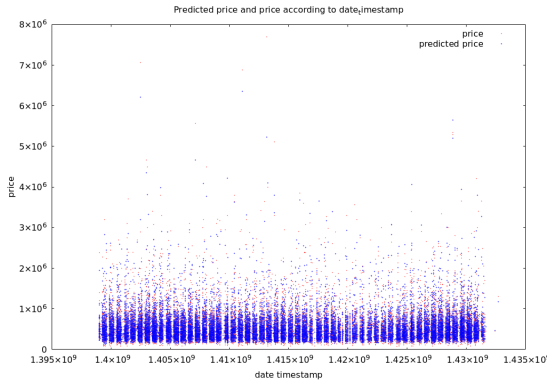

Figure 23: Sadly we don't have data going as far as 2008. Here timestamps don't really help the prediction it seems.

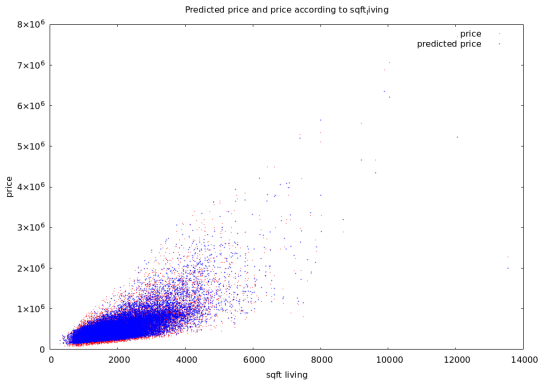

Figure 24: The living room's squared foots has a significant correlation with price. But is far from enough to explain it as the prices are widespread.

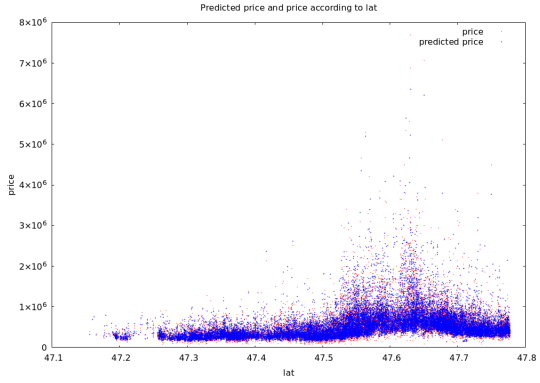

Figure 25: Prices from South to North

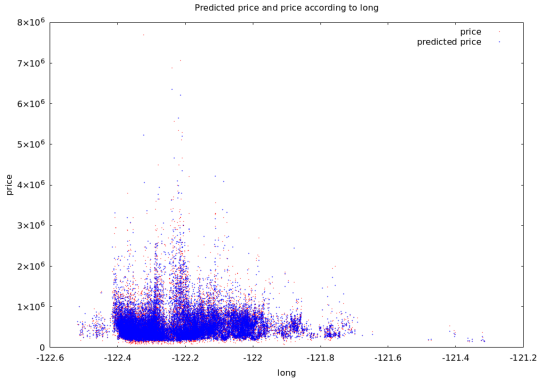

Figure 26: Prices from West to East

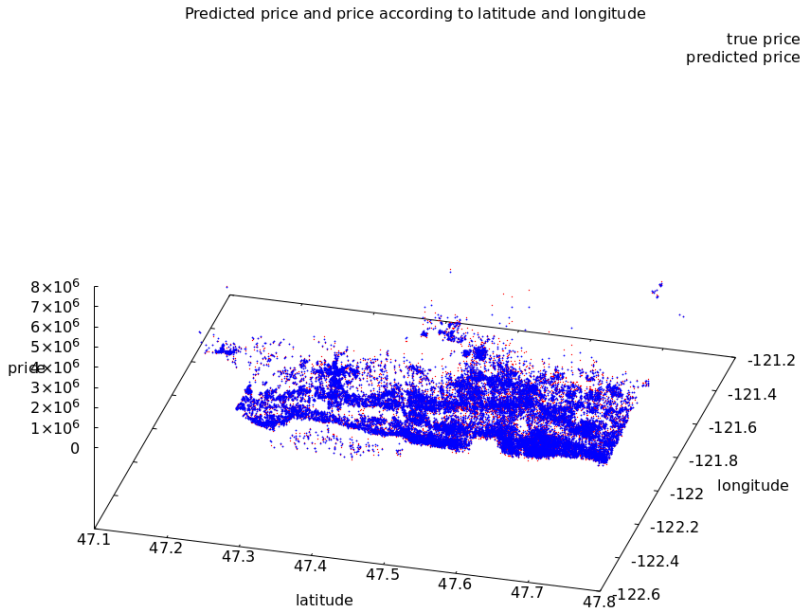

Figure 27: Latitude and longitude together. We can better see the most expensive neighbors in the North-West region.

2. j. d. Model interpretability via feature importance

One way to know which features are the most critical and decisive for the model to predict the true price is the technique of *feature importance*. This technique was first introduced with random forests in 2001 [23], but in 2018 a model-agnostic version was proposed [24], which can be used with regression models.

The process is simple:

1. Compute the accuracy of the model's predictions on the test data (I used R^2).
2. Take the first feature, shuffle its values across all the dataset's rows.
3. Compute the difference between the already computed accuracy of the model's predictions, minus the accuracy of the model's predictions on this sabotaged data.
4. Reset the data to its original form.
5. Repeat steps 3.) and 4.) a few times (I did 10 times) and do the average of the accuracies differences. Then save it alongside the feature's name or id.
6. Repeat steps 3.) to 5.) with the next feature, until you have done it with all of them.

The idea is that if your model had no to little accuracy difference while predicting using the normal data and while predicting with a feature being shuffled across all rows (hence its value being decorrelated from the rest), then it means that the feature does not inform the prediction much. On the other hand, if decorrelating the feature diminishes the accuracy, then it was certainly important to the model.

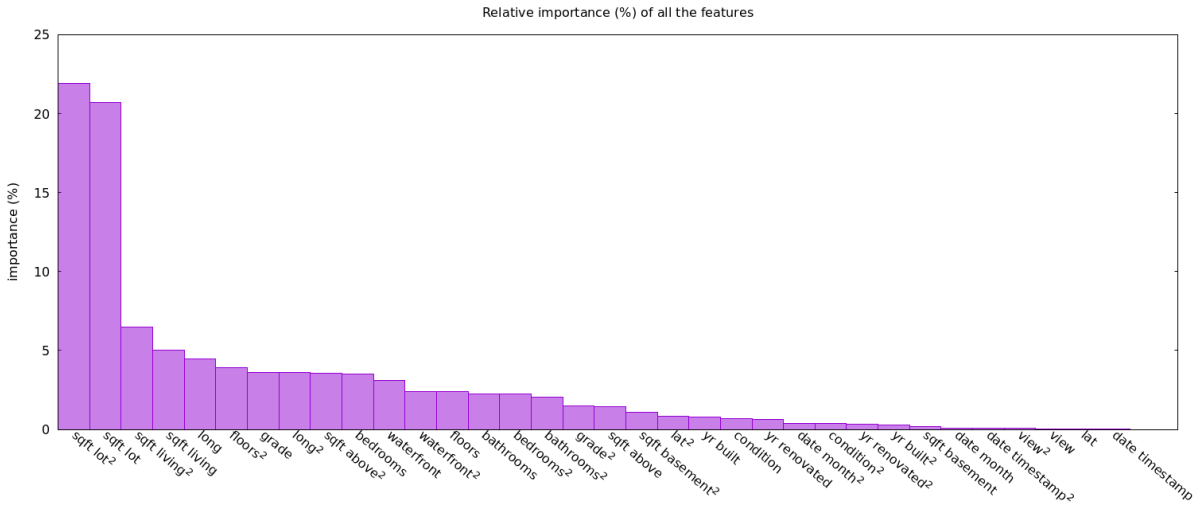

Figure 28: Features sorted by relative importance. Sum of all the feature's relative importance is 100%.

In this chart we can see that the lot's square foots and the living room square foot's are the most important features for the model to get high R^2. But the date or whether the house has a view of the bay didn't matter that much relative to all the other features.

Although I computed them on the whole dataset, my model has somewhat differently ordered relative importance scores for each feature than the Keras workflow's author [2]. Maybe it depends a lot on the minima found, but it would seem odd as I can't imagine models ordering features importance that differently from one another both having a similarly high R^2 score. Or maybe it depends mostly on the data, as some parts of the dataset may have a price explained abnormally better by certain features, and it is quite possible we haven't done that computation on the same data.

It takes a long time to compute these importance scores (around 15 minutes on my machine), but I'll try to investigate.

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 (almost as simple as I can do with Rust for most parts of the user-facing code).

Additionally to every feature 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.

I also implemented a fully unit-tested custom matrices backend, and a variation of it sped-up by a factor ~ 2.5 using CPU parallelism with the `rayon` library. Both of those custom backends are still not as fast as aforementioned libraries. My implementation may be sound time complexity-wise, it is not cache-optimized unlike the two libraries, and it is not building a lazy computation graph like the most optimized C++ libraries on the CPU do. I don't believe it would be worth the effort implementing that as the easiest logical next step for performance improvement would be running on the GPU.

The framework is open-sourced on Github with 60 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 engineers 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 [25], (or at least plugging an existing one if it ends up being too difficult).

After that, I would like to explore CNNs and dimensionality reduction layers for image and pattern recognition, and then RNNs for time series prediction or text translation. But since these problems would require significantly higher dimensional data, I prefer to wait until I make the framework GPU-bound.

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 versatile and highly dynamic tools allow small iterations and greater ease of use, making cutting-edge research faster. But they present a compromise over stability, maintainability, and raw performance.

In order for a ML workflow to be used at scale, low-level languages and GPU code is often used to make SDKs or libraries, 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 accessible high-level abstractions for concurrency, architecture, data structures and functional programming, which may very well be useful for building and maintaining large SDKs in the ML industry.

Rust aims to fill this niche for a low-level language feeling high-level, with its "Zero-Cost abstractions" philosophy [26], rich ecosystem of libraries [27], 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 [28], and even more in the graphics programming community, which gave birth to fast libraries for linear algebra, Cuda and OpenCL interoperability, 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's best practices do not resemble classic Object Oriented programming ones. It is considered by many as a language with a steep learning curve. And it is absolutely the language that took me the most work to become somewhat confident with. Other skills are required for building an ML library in Rust, such as good codebase planning, thoughtful library design, and a lot of back and forth API changes. In my opinion, this is why I initially struggled to find a polished, well documented and easy to use existing tool in Rust: It is just hard and slow to build.

So, even thought I made a footstep in the Rust ML ecosystem with this project, it is still very tiny, and we may be years apart from a tool as ergonomic as Pytorch or TF in Rust. But I believe the quest is well worth the effort, and others are working on it with a rich ecosystem of ML tools in Rust currently in the making [28].

But most importantly perhaps, I got to hack, experiment and build my own understanding.

"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, or depend on, the framework's `0.2.0` version.

5. a. Specifying the model as code

```
// Including all features from some CSV dataset
let mut dataset_spec = Dataset::from_csv("dataset/kc_house_data.csv");
dataset_spec
    .remove_features(&["id", "zipcode", "sqft_living15", "sqft_lot15"])
    // Add descriptive properties to the data features
    .add_opt_to("date", DateFormat("%Y%m%dT%H%M%S"))
    .add_opt_to("date", Not(&UsedInModel))
    // Add feature extraction/engineering instructions
    .add_opt_to("date", AddExtractedMonth)
    .add_opt_to("date", AddExtractedTimestamp)
    .add_opt_to(
        "yr_renovated",
        Mapped(
            MapSelector::Equal(0.0.into()),
            MapOp::ReplaceWith(MapValue::Feature("yr_built".to_string())),
        ),
    )
    .add_opt_to("price", Out)
    .add_opt(Log10.only(&["sqft_living", "sqft_above", "price"]))
    // inc_added_features makes sure the instruction is added to features
    // that were previously extracted during the pipeline
    .add_opt(AddSquared.except(&["price", "date"]).incl_added_features())
    .add_opt(FilterOutliers.except(&["date"]).incl_added_features())
    .add_opt(Normalized.except(&["date"]).incl_added_features());

// Building the layers
let h_size = dataset_spec.in_features_names().len() + 1;
let nh = 8;
let dropout = None;

let mut layers = vec![];
for i in 0..nh {
    layers.push(LayerSpec::from_options(&[
        OutSize(h_size),
        Activation(ReLU),
        Dropout(if i > 0 { dropout } else { None }),
        Optimizer(momentum()),
    ]));
}

let final_layer = LayerSpec::from_options(&[
    OutSize(1),
    Activation(Linear),
    Dropout(dropout),
    Optimizer(momentum()),
]);
```

```
// 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(500),
]);

// Saving it all to a JSON specification file
model.to_json_file("models/my_model.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
    .push(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 & all training metrics per folds & per epochs
// + joining all the predictions made on each fold during the 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(|fold, epoch, report| {
        println!("Perf report: {:2} {:4} {:#?}", fold, epoch, report)
    })
    .all_epochs_validation()
    .all_epochs_r2()
    .compute_best_model()
    .run(&model, &data);

// Saving the weights and biases of the best model
let best_model_params = kfold.take_best_model();
best_model_params.to_json(format!("models_stats/{}_best_params.json", config_name));

// 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 the predictions and evaluations (train/test losses, R²...)
data_and_preds.to_file("my_model_preds.csv");
model_eval.to_json_file("my_model_evals.json");
```

5. c. Computing relative importance

```
// Building the network from the specification
let mut network = model.to_network();

// Splitting the x and y
let (x_table, y_table) = data.random_order_in_out(&out_features);
let x = x_table.to_vectors();
let y = y_table.to_vectors();

// Loading the network's weights if necessary
let weights = NetworkParams::from_json("my_weights.json");
network.load_params(&weights);

// Predicting everything and computing the accuracy
let preds = network.predict_many(&x);
let ref_score = r2_score_matrix(&y, &preds);

let mut x_cp = x.clone();
let shuffles_count = 10;
let ncols = x_cp[0].len();
let mut means_list = Vec::new();

// For each feature
for c in 0..ncols {
    // Shuffling the feature's column 10 times
    let mut metric_list = Vec::new();
    for _ in 0..shuffles_count {
        shuffle_column(&mut x_cp, c);
        // Predicting everything and seeing if it worsens the model
        let preds = network.predict_many(&x_cp);
        let score = r2_score_matrix(&y, &preds);
        metric_list.push(ref_score - score);
        x_cp = x.clone();
    }
    // Doing the average for that feature
    means_list.push(avg_vector(&metric_list));
}

// Attaching the feature name and converting it all to percentages
let mut importance_rel = Vec::new();
let columns_names = x_table.get_columns_names();
let means_sum = means_list.iter().sum::<Scalar>();
for (mean, name) in means_list.iter().zip(columns_names.iter()) {
    importance_rel.push(((100.0 * mean) / means_sum, name));
}
// Sorting
importance_rel.sort_by(|a, b| b.0.partial_cmp(&a.0).unwrap());
```

5. d. K-folds detailed code

```
/// Running one fold in parallel
fn parallel_k_fold(
    &mut self,
    i: usize,
```

```
    model: &Model,
    data: &DataTable,
    // Mutexes in order to guarantee thread-safe
    // mutability
    // Arcs are thread-safe reference counted pointers
    // for the compiler to have memory safety guarantees
    validation_preds: &Arc<Mutex<DataTable>>,
    model_eval: &Arc<Mutex<ModelEvaluation>>,
    trained_models: &Arc<Mutex<Vec<Network>>>,
    k: usize,
) -> thread::JoinHandle<()> {
    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 all_epochs_r2 = self.all_epochs_r2;
    let reporter = self.real_time_reporter.clone();
    let trained_models = trained_models.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();

        // Split the data between validation and training
        let (train_table, validation) = data.split_k_folds(k, i);

        // Shuffle the validation and training set and split it between x and y
        let (validation_x_table, validation_y_table) =
            validation.random_order_in_out(&out_features);

        // Convert the validation set to vectors
        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 {
            // Train the model with the k-th folds except the i-th
            let train_loss = model.train_epoch(e, &mut network, &train_table,
id_column);

            // Predict all values in the i-th fold
            // It is costly and should be done only during the last epoch
            // and made optional for all the others in the future
            let loss_fn = model.loss.to_loss();
            let (preds, loss_avg, loss_std) = if e == model.epochs - 1 {
                network.predict_evaluate_many(&validation_x, &validation_y, &loss_fn)
            } else {
                (vec![], -1.0, -1.0)
            };

            // Compute the R2 scoreif it is the last epoch
            // (it would be very costly to do it every time)
            let r2 = if e == model.epochs - 1 || all_epochs_r2 {
```

```
                r2_score_matrix(&validation_y, &preds)
            } else {
                -1.0
            };

            // Build the benchmork of the model for that epoch
            // Useful for plotting the learning curve
            let eval = EpochEvaluation::new(train_loss, loss_avg, loss_std, r2);

            // Report the benchmark in real time if expected
            if let Some(reporter) = reporter.as_ref() {
                reporter.lock().unwrap()(i, e, eval.clone());
            }

            // Save the predictions if it is the last 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);
        }
        trained_models.lock().unwrap().push(network);
        model_eval.lock().unwrap().add_fold(fold_eval);
    });
    handle
}
```

5. e. SGD, Momentum and Adam

```
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. f. Dense layer and backpropagation

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

- My framework's source code on Github
- My King County House solution source code on Github
- The online documentation of my framework.

7. Bibliography

Here are some insightful videos, articles and papers I've read or cherry-picked information from:

[1] "kc_house_data," 2018. [Online]. Available: https://www.kaggle.com/datasets/shivachandel/kc-house-data

[2] Chavesfm, "DNN House Price R^2=0.88," *Kaggle*, 2021. [Online]. Available: https://www.kaggle.com/code/chavesfm/dnn-house-price-r-0-88/notebook

[3] O. Aflak, "Neural Network from scratch in Python - Towards Data Science," 2021. [Online]. Available: https://towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65

[4] A. Amini, "MIT Introduction to Deep Learning | 6.S191," 2023. [Online]. Available: https://www.youtube.com/watch?v=QDX-1M5Nj7s

[5] S. Lau, "Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning," 2018. [Online]. Available: https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1

[6] Tensorflow, "tf.keras.optimizers.schedules.InverseTimeDecay." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/InverseTimeDecay

[7] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, 2014. [Online]. Available: https://jmlr.csail.mit.edu/papers/volume15/srivastava14a/srivastava14a.pdf

[8] G. E. Hinton, "Improving neural networks by preventing co-adaptation of feature detectors," 2012. [Online]. Available: https://arxiv.org/abs/1207.0580

[9] U. Tewari, "Regularization — Understanding L1 and L2 regularization for Deep Learning," 2022. [Online]. Available: https://medium.com/analytics-vidhya/regularization-understanding-l1-and-l2-regularization-for-deep-learning-a7b9e4a409bf

[10] D. Godoy, "Gradient Descent, the Learning Rate, and the importance of Feature Scaling," 2022. [Online]. Available: https://towardsdatascience.com/gradient-descent-the-learning-rate-and-the-importance-of-feature-scaling-6c0b416596e1

[11] G. Aksu, C. O. Güzeller, and M. Eser, "The Effect of the Normalization Method Used in Different Sample Sizes on the Success of Artificial Neural Network Model," *Int. J. Assessment Tools Educ.*, pp. 170–192, 2019, doi: 10.21449/ijate.479404. [Online]. Available: https://dergipark.org.tr/en/download/article-file/682739

[12] J. Brownlee, "A Gentle Introduction to k-fold Cross-Validation," *Machinelearningmastery.com*, 2020. [Online]. Available: https://machinelearningmastery.com/k-fold-cross-validation/

[13] A. Cotter, O. Shamir, N. Srebro, and K. Sridharan, “Better Mini-Batch Algorithms via Accelerated Gradient Methods,” vol. 24, 2011, pp. 1647–1655. [Online]. Available: https://papers.nips.cc/paper/2011/file/b55ec28c52d5f6205684a473a2193564-Paper.pdf

[14] S. L. Smith, “Don't Decay the Learning Rate, Increase the Batch Size,” 2017. [Online]. Available: https://arxiv.org/abs/1711.00489

[15] S. Ruder, “An overview of gradient descent optimization algorithms,” 2016. [Online]. Available: https://arxiv.org/abs/1609.04747

[16] J. Brownlee, “A Gentle Introduction to the Rectified Linear Unit (ReLU),” *Machinelearningmastery.com*, 2020. [Online]. Available: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” vol. 25, 2012, pp. 1097–1105. [Online]. Available: http://books.nips.cc/papers/files/nips25/NIPS2012_0534.pdf

[18] X. Glorot, and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

[19] “InterQuartile Range (IQR).” [Online]. Available: https://sphweb.bumc.bu.edu/otlt/mph-modules/bs/bs704_summarizingdata/bs704_summarizingdata7.html

[20] D. P. Kingma, “Adam: A Method for Stochastic Optimization,” 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[21] S. J. Reddi, S. Kale, and S. Kumar, “On the Convergence of Adam and Beyond,” 2018. [Online]. Available: https://arxiv.org/pdf/1904.09237

[22] “Numeracy, Maths and Statistics - Academic Skills Kit.” [Online]. Available: https://www.ncl.ac.uk/webtemplate/ask-assets/external/maths-resources/statistics/regression-and-correlation/coefficient-of-determination-r-squared.html#Interpretation%20of%20the%20R^2%20value

[23] L. Breiman, “Random Forests,” *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001, doi: 10.1023/a:1010933404324. [Online]. Available: https://link.springer.com/content/pdf/10.1023%2FA%3A1010933404324.pdf

[24] A. Fisher, “All Models are Wrong, but Many are Useful: Learning a Variable's Importance by Studying an Entire Class of Prediction Models Simultaneously,” 2018. [Online]. Available: https://arxiv.org/abs/1801.01489

[25] “wgpu: portable graphics library for Rust.” [Online]. Available: https://wgpu.rs/

[26] Mike-Barber, “GitHub - mike-barber/rust-zero-cost-abstractions: Testing out a Zero Cost Abstraction in Rust compared to similar approaches in C# and Java.” [Online]. Available: https://github.com/mike-barber/rust-zero-cost-abstractions

[27] “crates.io: Rust Package Registry.” [Online]. Available: https://crates.io/

[28] “Index.” [Online]. Available: https://www.arewelearningyet.com/