# Measuring the magnetic interactions between a pair of atoms in a molecule

Zengxiang Zhao

December 2019

**Abstract**

This project will use machine learning to predict the magnetic interaction between atoms in the molecule. The datasets of this project come from Kaggle's competition. We will use Flux, a Julia ML library, to construct the ML structures. In this project, we have tried three sorts of models. At first, we will use 11 parameters to train the model 1, and we fond that the higher the target value, the more accurate the model could predict. Then we add more parameters(39 parameters in total) to train the model 2, we could get a better model. During the training process, changing the learning rate will make the model learn faster, and it's better to set a high learning rate at the beginning and a low level at the end stage of the training process to avoid passing the optimal point. Model 3 we build from scratch can't work as we expected.

**Index Terms:** Julia, machine learning(ML), Tensorflow, magnetic interaction, Flux.

# I INTRODUCTION

The Computer plays a more and more important role in drug design, such as predicting the interaction between protein and its target[1] and elucidate the relation without synthesis and purification of proteins[2]. By using machine learning we could unearth more information from the data. Just as the web[3] said machine learning is a numerical programming language problem, and Julia is an excellent substrate to be competent. For example, Julia's mathematical syntax makes it an ideal way to express algorithms just as they are written in papers, and Julia's native GPU and Automatic Differentiation support boost the computational ability [4]. The data of this project is from Kaggle's competition: Predicting Molecular Properties[5]. We will use machine learning implemented by Julia to predict the scalar-coupling-constant between atom pairs in molecules, given the two atom types.

TensorFlow, designed by Google, is an open-source library to develop and train ML models easily. At first, I want to use the TensorFlow framework[6] to design the machine learning structure. But my computer is not compatible whit Julia *tensorflow.jl* cause the version of Python is 3.7 instead of Python2.7 or Python3.6. While even though I change the Python version, it still doesn't work as shown in Figure 1.

Thus I will use the Flux, the Julia machine learning library is written in numerical computing language Julia, to create an ML structure. It can use the full power of the Julia language[7], such as using Julia's simple mathematical syntax and create "dynamic"

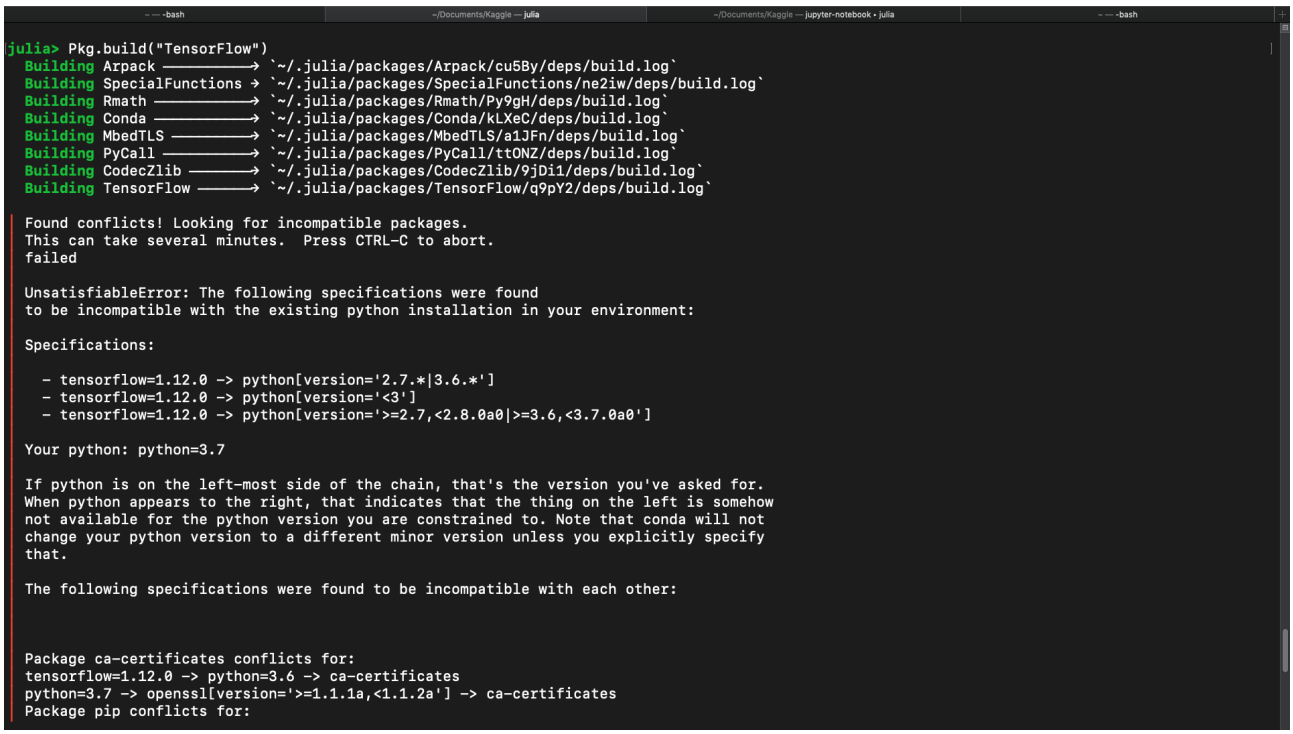


Figure 1: Failing installing the tensorflow.jl

frameworks to make full use of the compiler features of Julia[8]. While unlike Julia, TensorFlow.jl has not the full-capacity to directly leverage some of the benefits of the language and its ecosystem[6]. Three pillars set Flux apart among ML systems: simplicity, hackability, and underlying compiler technology [9].

# II SOFTWARE

To finish this project, The following software will be used:

1. Jupyter-notebook.
2. Julia.

   Three packages are installed:

   - DataFrames : to show the data of this project.[10]
   - Flux : Julia ML library
   - MLDataUtils : to split the data into train and test[11].
3. Overleaf: an online LaTeX editor.[12]

# III WORK SHOW

1. **Searching Data:** This project is about chemistry. Thus I searched the Kaggle and found a competition about predicting the interactions between a pair of atoms in a molecule[5]. I will use the julia ML to compute the interactions between atoms.
2. **Installing ML library:** I searched the web to decide which library is available for ML in Julia. And I found TensorFlow and Flux can be competent. At first, I want to use TensorFlow as the library to implement the ML structures as I've used it before. But it can't be installed correctly on my computer as shown in figure 1. Thus I switch to use Flux as the ML library.
3. **Learning Flux:** Flux is a new Julia library and there is only one year old [9]. The references about Flux are only the official manuals[4] and a few ML examples are available. After implementing a simple ML structure by Julia and reading the ML examples, I finally know the main idea of Julia ML model. Thus in

this project I use both the model of Flux and the model created from scratch to do machine learning.

4. **Organising the data :** there are total seven datasets will be used in this project, which are:

   (a) `"dipole_moments.csv"`

   (b) `"magnetic_shielding_tensors.csv"`

   (c) `"mulliken_charges.csv"`

   (d) `"potential_energy.csv"`

   (e) `"scalar_coupling_contributions.csv"`

   (f) `"structures.csv"`

   (g) `"train.csv"`

   At first, we will just use the dataset consisting of train.csv and structures.csv to do machine learning. There are eleven parameters in this dataset, and five of them will be translated into one hot batch. In this dataset, we eliminate the parameter "molecule names". Because there are 85003 unique molecule names in the dataset($df_{22}$). Then 85003 rows will be added to the training data if we use one hot batch. And the training matrix will be bigger than 850004658147, which My computer can't deal with. Thus I delete this column in the dataset.

   On the other hand, we also combine all these seven datasets into a complete datasets that contain 39 parameters.

5. **Constructing the ML structure:** At first, I want to borrow the idea from the kernel of a Kaggle competition [13] and translate this kernel into Julia language with some modification. But I found creating a Julia ML model is easier. We create three Julia ML models shown as follows:

```
m = Chain(
     Dense(76,1),

)

loss(x, y) = Flux.mse(m(x_train),
    y_train)
dataset = repeated((x_train,
    y_train), 20)
evalcb = () -> @show(loss(x_train,
     y_train))
opt = ADAM()
Flux.train!(loss, params(m),
    dataset, opt, cb = throttle(
    evalcb, 2))
```

Listing 1: Model-1

```
mc = Chain(
     Dense(104,1),

)
loss(x, y) = Flux.mse(mc(x), y)
dataset_complete = repeated((
    xc_train, yc_train), 20)
evalcb_complete = () -> @show(loss
    (xc_train, yc_train))
opt_complete = ADAM(0.5)
Flux.train!(loss, params(mc),
    dataset_complete, opt_complete,
     cb = throttle(evalcb_complete,
     2))
```

Listing 2: Model-2

```
predict(x) = W*x .+ b
meansquarederror(y  , y) = sum((
    y   .- y).^2)/size(y, 2)
loss(x, y) = meansquarederror(
    predict(x), y)
   = 0.1
   = Params([W, b])
for i = 1:10
  g = gradient(() -> loss(xc_train
    , yc_train),   )
  for x in
    update!(x, -g[x]*   )
  end
  @show loss(xc_train, yc_train)
end
```

Listing 3: Creating the model from scratch. [14]

6. **MLDataUtils package :** We use MLDataUtils package to split the dataset into train set and test set. The codes are shown in the following:

```
Xs, Ys = shuffleobs((x, y))
(x_train,y_train), (x_test,y_test)
    = splitobs((Xs, Ys); at = 0.1)
```

Listing 4: Usage of MLDataUtils

The results we get, x-train,y-train,x-test,y-test, are just lazy SubArrays instead of the actual arrays. This process will delay the actual access of the data until the data is needed.

# IV RESULTS

**Process of ML:**

1. Combine the dataframes into datarame A.
2. Obtain datarame B from deleting the one-hot batch columns in the dataframe A .
3. Create one-hot batch: generating one hot batch of `:atom_index_0,:atom_index_1, :type,:atom_0,:atom_1`
4. Change the dataframe B into a Matrix C, and transpose the matrix to get C'.
5. Concatenate the transposed matrix C' and the one-hot batch to get the $x$.
6. Get the target value $y$
7. Split the data (x,y) into `(x_train,y_train)` and `(x_test,y_test)`
8. Define the ML model and train the ML model.

All the following results are shown in the Jupyter Julia notebook.

1. Results of Model 1: Changing the learning rate and repeated times of data could improve the model. The higher the value is, the better result we can get from the model. The comparison of $predict(x)$ of model 1 and target value $y$ is shown in the following:

```
Output:
# Check the model using train data
#  predict(x)                  y
(4.5997868f0 (tracked), 8.1091)
(3.164126f0 (tracked), -1.013)
(7.1762743f0 (tracked), 8.06721)
(1.6722097f0 (tracked), -1.20206)
(4.7480025f0 (tracked), -0.164238)
(3.7005816f0 (tracked), 1.80993)
(2.3056245f0 (tracked), -1.0258)
(2.331396f0 (tracked), -0.415738)
(91.20658f0 (tracked), 103.367)
(4.449092f0 (tracked), 1.6762)

# Check the model using test data
#  predict(x)                  y
(5.571581f0 (tracked), 1.95463)
(6.6052094f0 (tracked), 0.198646)
(5.966392f0 (tracked), 3.1511)
(1.3568573f0 (tracked), -1.81079)
(5.3730454f0 (tracked), 0.839303)
(4.6628094f0 (tracked), 1.04344)
(1.42906f0 (tracked), 5.62805)
(1.297679f0 (tracked), -1.00772)
(91.41441f0 (tracked), 91.2698)
(4.2279286f0 (tracked), 0.899873)
```

Listing 5: Results of model 1

2. Results of Model 2: Adding more parameters in the training data could get a better model.

```
Output:
# Check the model using train data
#  predict(x)                  y
(3.6611907f0 (tracked), 6.93372)
(6.166321f0 (tracked), 7.0001)
(2.094234f0 (tracked), 3.70842)
(-4.581895f0 (tracked), -2.54874)
(-20.302265f0 (tracked), -17.8543)
(6.0142183f0 (tracked), 9.91444)
(-0.90044427f0 (tracked), 3.45232)
(-15.803331f0 (tracked), -12.7186)
(-0.028750844f0 (tracked),
    1.43408)
(-3.6505148f0 (tracked), 0.073232)

# Check the model using test data
#  predict(x)                  y
(3.1146507f0 (tracked), 5.08376)
(-2.765433f0 (tracked), -2.76404)
(-8.194154f0 (tracked), -4.37406)
(-2.704211f0 (tracked), -0.399917)
(-1.329688f0 (tracked), 1.61685)
(95.235214f0 (tracked), 99.06)
(-4.1642375f0 (tracked),
    -0.712649)
(-1.1586939f0 (tracked), 0.840197)
(81.48726f0 (tracked), 86.1891)
(79.35697f0 (tracked), 83.8421)
```

Listing 6: Results of model 2

3. Results of model 3: Model 3 couldn't work as the loss getting bigger and bigger. The reason may be that the model is unable to optimize the parameters.

```
#Output of training
loss(xc_train, yc_train) =
    2.4268499601377035e13 (tracked)
```

```
loss(xc_train, yc_train) =
    4.023245581459924e22 (tracked)
loss(xc_train, yc_train) =
    6.67009572677111195e31 (tracked)
loss(xc_train, yc_train) =
    1.1058281558472694e41 (tracked)
loss(xc_train, yc_train) =
    1.8333408705593197e50 (tracked)
loss(xc_train, yc_train) =
    3.03947654967295e59 (tracked)
loss(xc_train, yc_train) =
    5.039116208211317e68 (tracked)
loss(xc_train, yc_train) =
    8.354297769657191e77 (tracked)
loss(xc_train, yc_train) =
    1.3850502417540682e87 (tracked)
loss(xc_train, yc_train) =
    2.296260230453465e96 (tracked)
```

Listing 7: Results of model 3

# V CONCLUSION

We use Flux as the machine learning library to predict the magnetic interaction between atoms in the molecule. Flux is easy to implement the ML model as it is written in Julia's language and make full use of the power of Julia. In this project, we have tried three models using two different training data. One has 11 parameters, and another has 39 parameters. We've found that more parameters could improve the model accuracy and the predicted value is more accurate when the target value is high. The model we build from scratch can't work as we expected. The reason may be that the model is unable to update the parameters correctly.

# References


[1] J. Zeng, "Mini-review: computational structure-based design of inhibitors that target protein surfaces," *Combinatorial Chemistry & High Throughput Screening*, vol. 3, no. 5, pp. 355–362, 2000.

[2] J.-H. Lin, A. L. Perryman, J. R. Schames, and J. A. McCammon, "Computational drug design accommodating receptor flexibility: the relaxed complex scheme," *Journal of the American Chemical Society*, vol. 124, no. 20, pp. 5632–5633, 2002.

[3] M. Innes, S. Karpinski, V. Shah, D. Barber, P. Saito Stenetorp, T. Besard, J. Bradbury, V. Churavy, S. Danisch, A. Edelman *et al.*, "On machine learning and programming languages." Association for Computing Machinery (ACM), 2018.

[4] "Machine learning and artificial." [Online]. Available: https://juliacomputing.com/domains/ml-and-ai.html

[5] "Predicting molecular properties." [Online]. Available: https://www.kaggle.com/c/champs-scalar-coupling/data

[6] J. Malmaud and L. White, "Tensorflow.jl: An idiomatic julia front end for tensorflow." *J. Open Source Software*, vol. 3, no. 31, p. 1002, 2018.

[7] "Flux: The julia machine learning library." [Online]. Available: https://fluxml.ai/Flux.jl/stable/

[8] M. Innes, "Flux: Elegant machine learning with julia." *J. Open Source Software*, vol. 3, no. 25, p. 602, 2018.

[9] M. Innes, E. Saba, K. Fischer, D. Gandhi, M. C. Rudilosso, N. M. Joy, T. Karmali, A. P. Singh, and V. Shah, "Fashionable modelling with flux," *arXiv preprint arXiv:1811.01457*, 2018.

[10] "Dataframes documents." [Online]. Available: https://juliadata.github.io/DataFrames.jl/stable

[11] "Mldatautils documents." [Online]. Available: https://mldatautilsjl.readthedocs.io/en/latest/index.html

[12] "Overleaf documents." [Online]. Available: https://www.overleaf.com/learn

[13] "Keras neural net for champs." [Online]. Available: https://www.kaggle.com/todnewman/keras-neural-net-for-champs

[14] "Mlp on housing data (low level api)." [Online]. Available: https://github.com/FluxML/model-zoo/blob/master/other/housing/housing.jl