Predicting Molecular Activity Using Deep Learning in TensorFlow

This work provides a solution to the Merck Molecular Activity Kaggle Challenge using deep learning with the TensorFlow library.

1. Problem Description

Predicting the activity of a novel compound is a real challenge encountered in pharmaceutical companies when developing new drugs. During the drug discovery process, it is standard practice to scan a large library of compounds to test their biological activity towards both the intended target molecule as well as peripheral off-target molecules. It is desirable for a compound to be highly specific in order to have high efficacy and minimize side effects – that is, it should be highly active against the target molecule while showing low or no activity towards off-target molecules. This process can be very time-consuming and expensive. It is therefore desirable to quantitatively predict such biological activity between compounds and both target and off-target molecules. Such predictions can help prioritize experiments and reduce the experimental work that needs to be performed. Indeed, such predictions are called Quantitative structure−activity relationships (QSAR) and are very commonly used in the pharmaceutical industry.

The QSAR method has the following characteristics:

1) The data set in a pharmaceutical environment usually involves a large library of compounds. For example, each target is tested against over 20,000 compounds in this Merck data challenge.

2) Each compound is usually represented by a list of features or fingerprint descriptors. These fingerprints typically describe the content, chemistry, and molecular topology of the molecule and are encoded by a feature vector. For example, content and chemistry can include descriptors for the number of carbon and nitrogen atoms, hydrogen-bonding donors and acceptors, charges, polar and nonpolar atoms, and functional groups (ketone, carboxylic acid, etc). The topology includes which atoms are bonded to which other atoms within a specific radius (i.e. a subgraph of the molecular graph). The Merck challenge as described on the Kaggle does not give the specific fingerprints used, only the feature vectors as raw data.

3) Each set of compounds may involve a total number of thousands of fingerprint descriptors. Therefore, the vectors describing each compound are sparse and only a small proportion of them are nonzero.

4) These descriptors are not all independent. Depending on the structure of the compound, strong correlations may exist between different descriptors.

2. Objective & Data

The objective of this Merck data challenge is to identify the best statistical techniques for predicting biological activities of compounds against specific molecules given the numerical descriptors generated from the chemical structures of these compounds.

The data provided by Merck is based on 15 target molecules and over 20,000 compounds for each target. For each target molecule, each row of the data corresponds to a compound and contains

descriptors derived from that compound's chemical structure. Activity between the target molecule and each compound is provided in the training data and is the target for prediction in the test data.

3. Evaluation Metric

Predictions for activity will be evaluated using the correlation coefficient R^2, averaged over the 15 data sets. For each data set,

$$r_s^2 = \frac{[\sum_{i=1}^{N_s}(x_i - \bar{x})(y_i - \bar{y})]^2}{\sum_{i=1}^{N_s}(x_i - \bar{x})^2 \sum_{i=1}^{N_s}(y_i - \bar{y})^2}$$

where x is the known activity, $\bar{x}$ is the mean of the known activity, y is the predicted activity, $\bar{y}$ is the mean of the predicted activity, and N_s is the number of molecules in data set s. An R^2 of 1 indicates the predictions perfectly fit the data.

4. Approaches and Solutions

Given that the target variable activity is continuous, and each dataset contains tens of thousands of entries, this is a regression problem and I will build neural networks (NN) using the TensorFlow library in Python to make predictions. The cost function is defined as mean squared error or MSE in this work.

Below is a summary of factors considered in this work:

4.1 One model for each target molecule or for all target molecules

Theoretically, merging the data for all 15 molecules and training a universal model will be ideal since a much larger training data will be used and the model may perform better. However, in this particular problem, these 15 target molecules can be very different from each other in both structures and activities. Thus, it is not immediately intuitive how data for one target molecule may be applicable for training and prediction on another molecule. Given the limit of computational power I can access and desired timeframe for completion of this work, one model for each target molecule may be preferred. Alternatively, we can attempt to train one good model and apply it over all 15 datasets. ***As a showcase, only the model for the first target molecule is trained and optimized in this work.***

4.2 Data preprocessing

4.2.1 Target

The training data has 37k entries among which over 20k have the same activity value 4.30 and the rest between 4.30 and 8.5 (Figure 1).

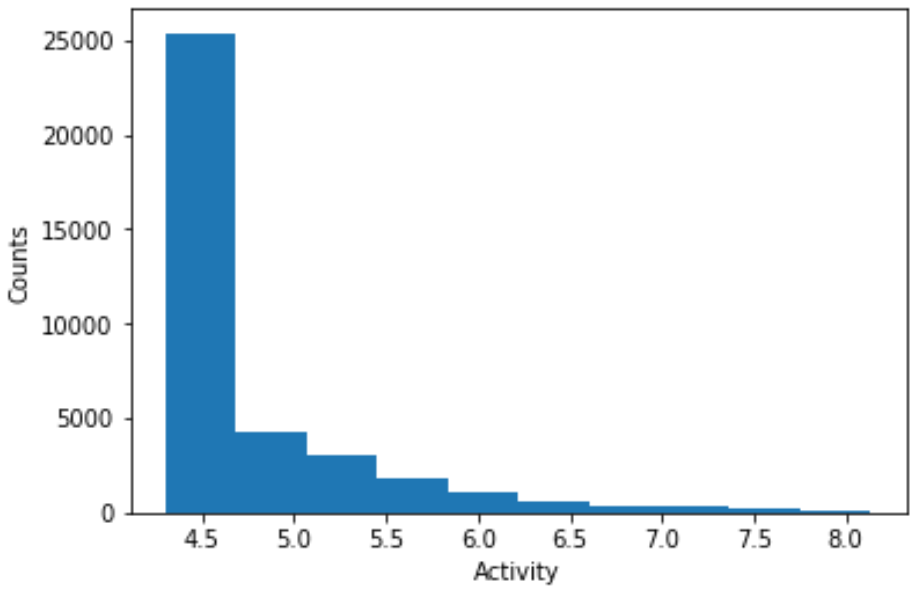

Figure 1. Histogram of activity values in the training data for molecule 1.

This almost certainly comes from a threshold encountered during the data collection process. For example, any data below 4.30 is not collected or mapped to 4.30 due to experimental limitations. Thus, a majority of the target data is truncated and the NN may have problems where it ends up predicting the outputs to be one single value. The following preprocessing steps are tried:

1) no preprocessing
2) remove all entries with a target value of 4.30
3) randomly remove 90% of all entries with a target value of 4.30

Among them, 1) appeared to give the highest and most consistent R^2 value, probably related to having the largest amount of input data fed into the NN (Figure 2). Thus, no preprocessing of the target y is done. However, the optimizer clearly has a difficult time with the dataset, which I will touch upon in section 4.3.5.

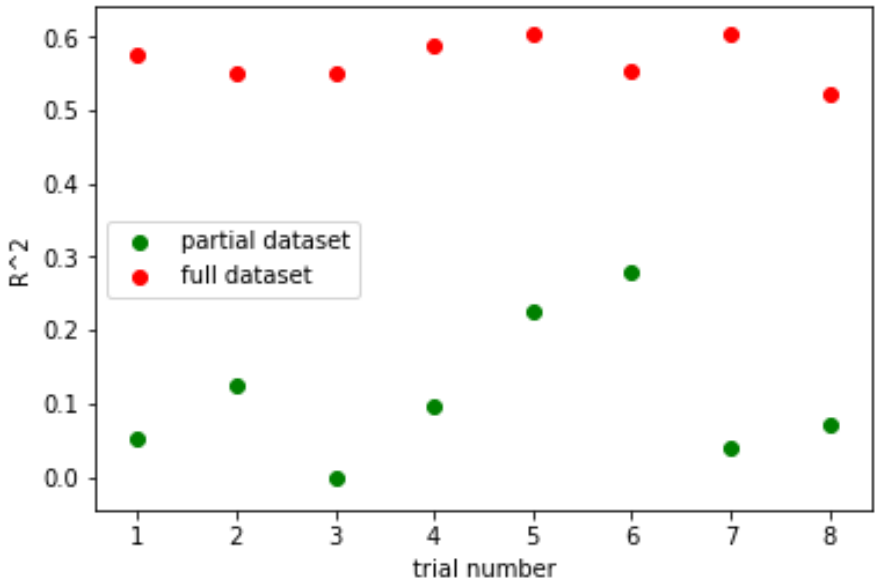

Figure 2. R^2 value for 8 runs of the same neural network with the full training dataset or partial dataset with all entries with a target value of 4.30 removed.

4.2.2 Defining the Training and Testing Data

Unfortunately, the true validation data for this Kaggle challenge has been taken off-line and submissions on the website have been turned off. Instead, I generated testing data by splitting the training data. I chose to use an 80% training to 20% testing split, which gives approximately 29.6k entries for training and 7.4k for testing.

4.2.3 Number of features

There are 9491 features for each entry and most feature vectors have a large number of zeros. The following preprocessing steps are tried in order to remove trivial information and speed up the training process of the model:

1) no preprocessing
2) filter out features with a variance below a threshold. For example, a threshold of 5e-4 will remove 3.31% of the features.
3) use principal dimensionality reduction (PCA) to reduce the dimension (for example, to 6000 instead of 9491 features)

It turned out that both 1) and 2) gives comparable R^2 while 3) gives lower R^2 values. Thus in the following results, the features are not preprocessed.

4.2.4 Descriptors

All descriptors are non-negative integers. Options for the descriptor transformations are:

1) no transformation
2) logarithmic transformation, i.e. $y = \log(x + 1)$
3) binary transformation, i.e. $y = 1$ if $x > 0$, otherwise $y = 0$.

Figure 3 shows how the cost function change with number of epochs for the above three different processing steps. It is obvious that binary or logarithmic transformation of the descriptors x will decrease the cost function to below 0.2 with less oscillation over the training period. **It is important to mention that I observed when the cost function is above 0.2 the R^2 is always close to 0.**

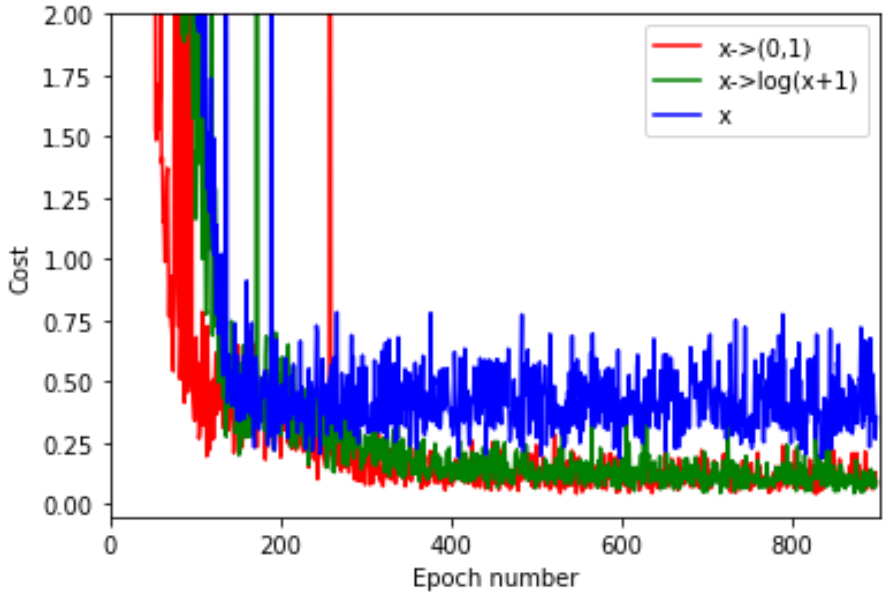

Figure 3. Change of cost function with increasing epoch number for three different x preprocessing.

Comparable R^2 values (around 0.6) are achieved using logarithmic and binary transformation of descriptors x. The logarithmic transformation of descriptor x is used for the rest of the training process.

4.3 Parameters of the neural network

4.3.1 Activation function

The practical role of activation functions is to enable NN to model complex, nonlinear functions. Without nonlinear activation functions, a NN will be equal to a linear function. In this problem, ReLu and Sigmoid are tried as activation functions. ReLu performs better than Sigmoid and is used in all hidden layers in this work, as the cost function doesn't converge when using Sigmoid activation function. This may be because the Sigmoid activation function confines all output between 0 and 1 and thus dampens the training ability of the NN for this particular problem.

Linear activation is used for the output layer, which is a common practice for training NN for regression problems.

4.3.2 Weight and biases

Given that the cost function easily goes to local minima, it's also important to start with optimized weight and biases. Our optimized weights and biases are random values from a truncated normal distribution with a mean of 0 and standard deviation of 1. Values who is outside of [-2, 2] are dropped and re-picked – this step is very important for getting consistent results.

4.3.3 Architecture of neural network

Every NN has at minimum three layers: input, hidden and output. For input there is exactly one input layer, and its number of neurons is uniquely determined by the shape of the training set. For the regression in this problem, NN has only one output layer with exactly one node and linear function as its activation function.

Determining the number of hidden layers is critical. While starting with one hidden layer is reasonable for a variety of problems, a NN with one hidden layer here gives R^2 value of around 0 or the meaningless value NaN (Not a Number) because all predicted target values are nearly exactly or exactly identical. Increasing the number of hidden layers to two without optimizing hyperparameters can give R^2 of around 0.56. Therefore, NN with at least two hidden layers are preferred. For the remainder of this work two hidden layers will be used.

As to the number of neurons in each hidden layer, the most common rule-of-thumb is 'the optimal size of the hidden layer is usually between the size of the input and size of the output layers'. Figure 4 compares the R^2 value for 4 different NN with different number of nodes in the hidden layers over eight separate training trials trained over 300 epochs with a sample size of 300 data points per batch. The NN with 5 and 2 nodes in the first and second hidden layers respectively has similar maximum R^2 value (~ 0.6) as the other three NN with more nodes, but it has three trials with much lower R^2 which indicates the instability of using a NN with a small number of nodes to model the data. The NN with 50 and 25 nodes in the first and second hidden layer shows the most consistent result across the 8 trials, with an average R^2 of 0.568 and a standard deviation of 0.027. The largest NN with 100 and 50 nodes in the first and second hidden layer also shows inconsistency among the 8 trials, with 1 meaningless R^2 and 1 R^2 close to 0. This is probably because 300 epochs is not enough for the larger NN to reach its global minimum consistently. Indeed, the bigger the NN, the more epochs it typically needs to run in order to get consistently decent R^2. Ideally, we want to run a NN as long as the computational power allows until we can see the optimizer converge on a solution.

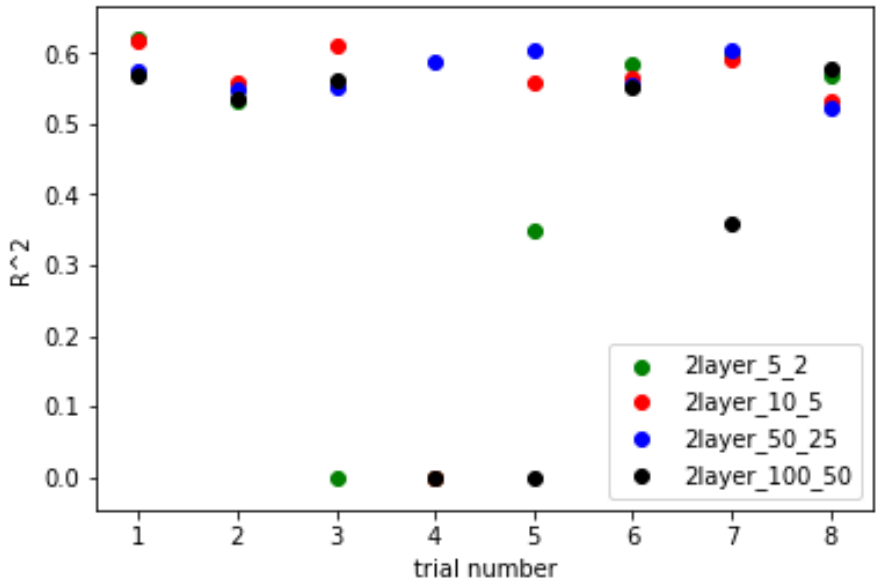

Figure 4. R^2 value in different trials for NN with 2 hidden layers and different node numbers ((5, 2), (10, 5), (50, 25), (100, 50) for the first and second hidden layer respectively.) 300 epochs are run for each NN.

4.3.4 Dropout

Dropout is a common strategy used for preventing overfitting in a NN. Optimized results in this work are achieved with 25% neural nodes dropped out randomly in the hidden layer and no dropout in the output layer.

4.3.5 Optimizer & Learning Rate

The Adam optimizer turned out to be the best optimizer. Stochastic gradient descent was initially used, but most times the cost function will be trapped in local minima (0.45 for instance) and decreasing the learning rate and adding momentum did not help much. After switching the optimizer to the Adam (adaptive moment estimation) optimizer, the cost function started to decrease to less than 1 and converge. The Adam optimizer computes adaptive learning rates for each parameter and keeps an exponentially decaying average of past gradient. It is usually the go-to optimizer for NN with input data as sparse matrix.

However, the Adam optimizer can still be trapped in local minimum. For example, Figure 5 shows the change of cost function and R^2 value along with increasing epoch number for a NN with two hidden layers with 1000 and 500 nodes respectively. At epoch 865, the cost does a small hop from below 0.2 to above 0.4 into a sharp local minimum that it never recovers from. The R^2 is very bad afterwards, while it was much better before epoch 865. The function landscape as defined by the data is likely to encompass a lot of very sharp local minima, since we often see training trials get trapped at around the 0.4 - 0.5 MSE with bad R^2 (see Figure 4 for some examples). This hopping phenomenon happens randomly during the optimization trajectories and is hard to control. Thus, for each optimized model, I usually run it multiple times to get consistent results.

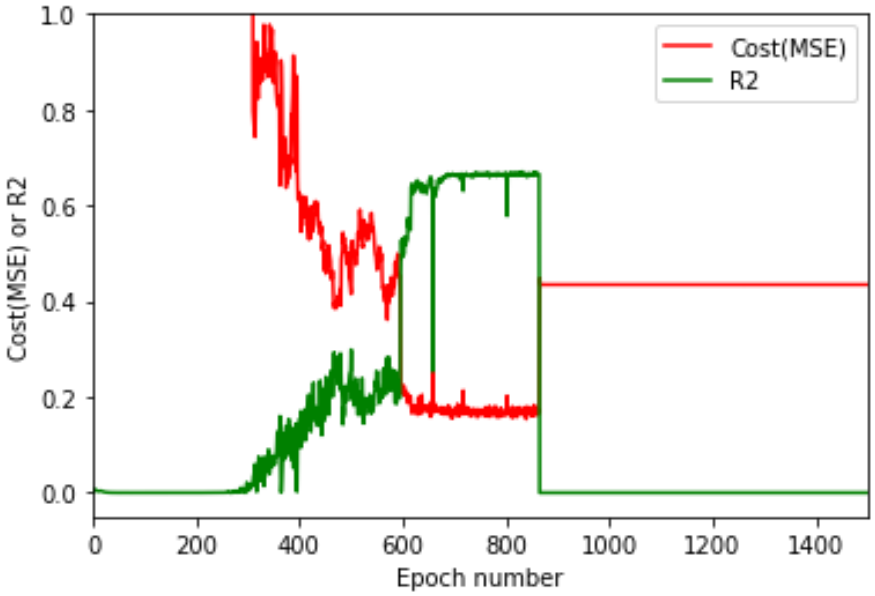

Figure 5. Change of cost function and R^2 value with epoch number. The NN has 2 hidden layers with 1000 and 500 nodes respectively. All other parameters of the NN is optimized as shown in this report.

The output of the NN is very sensitive to the initial learning rate of the gradient descent, especially when using stochastic gradient descent as the optimizer. I started with a learning rate of 0.05 and noticed that the cost function either diverges or quickly goes beyond the maximum floating point number, giving NaN. Decreasing the learning rate will decrease the cost function, however, the optimizer can be easily trapped in local minimum (24.085, 22.220, or 14.683, for instance). Only once the learning rate decreased to 0.001 do I start to see consistent output of cost and R^2 value over 0.1. Thus, the learning rate is kept at 0.001.

5. Summary

In conclusion, the optimized NN that gives the highest R^2 value has the following characters:

- All training data and all descriptors are used without any filtering process.
- Logarithmic transformation of descriptors x and no preprocessing of target y.
- The NN has two hidden layers. The first and second hidden layers have 50 and 25 neurons respectively.
- ReLu is used as the activation function for all hidden layers. Linear activation function is used for the output layer.
- The learning rate is kept at 0.001.
- Weights and biases are random values from a truncated normal distribution with a mean of 0 and standard deviation of 1. Values who is outside of [-2, 2] are dropped and re-picked.
- Adam optimizer is used.
- A dropout rate of 25% is used for all hidden layers and no dropout is used in the output layer.
- Batch size is 300 and epoch number is 900.

The highest R^2 value on the test data (which is 20% split from the training data) is 0.568 +/- 0.027. The Kaggle leadership board shows a top average R^2 score of 0.494 for all 15 molecules. Compared to that result, the R^2 value from the NN trained in this work is quite satisfactory.