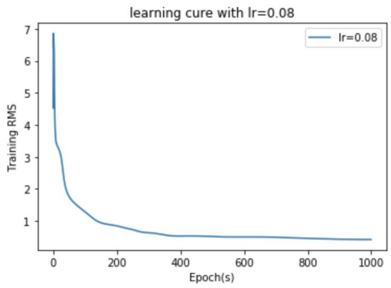
HW1 report

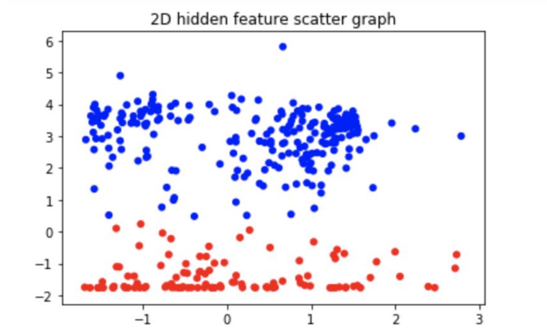
written on: 2019.11.06
last modified: 2019.11.07
author: 0712238 林彥彤, Maxwill Lin

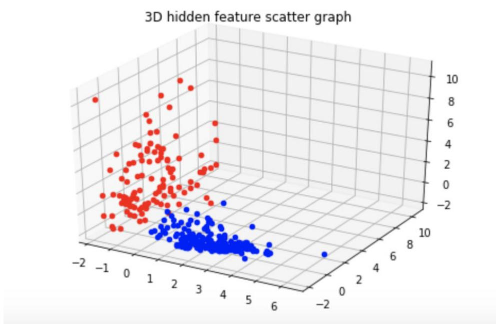
Answer to the problems:

- solutions and experiments are in pdf report of ipython notebook

```
train_RMS =  0.654207431054541
 test_RMS =  1.1595441303093315
: 1000
```

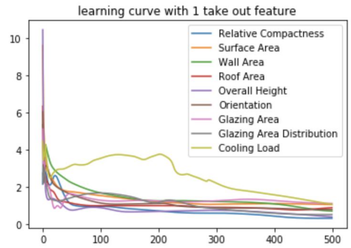


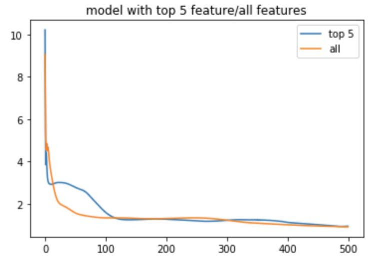




- more clarification of problem 1-c(experiments):
  To find the important factors, I think removing one at once to see how much does the RMS error change is a good way if the given data features are independent with each other(which can be checked by other methods like covariance matrix).
  The more the corresponding RMS rises, the more important should the factor be.
  This idea is implemented with ipython notebook files.
  And the final top 5 influential feature is quite robust and performed similarly with all-feature model

```
importance ranking: (measurement = avg test RMS over 10 experiments)
Glazing Area Distribution 1.9636342650841616
Glazing Area 1.5541616688745346
Cooling Load 1.453779166166917
Roof Area 1.2235254902579185
Surface Area 1.1916754302488417
Wall Area 1.1290687148035499
Overall Height 1.1114369513207476
Relative Compactness 1.1095003688360312
Orientation 0.8525229558428814
```





Discussion:

- Data Preprocessing:
  - regression:
    - drop one-hot features and y(heating load)
    - make one-hot features(orientation, etc.) (considered as a feature n-d vector of unit norm)
    - normalize non-one-hot features(important if there is one-hot feature)
    - combine
  - classification:
    - drop y is enough
    - the features seem already normalized
  - implementation : pandas + numpy
  - dataframe.drop([comlum names], axis=1)
  - dataframe.values
  - implementation of one-hot encoding and normalization
  - np.c_[a,b,c]

- Model Topology Design:
  - Overview:
    - follow my intuition at first and try some different models
    - not complicated task + small training data size
      => should use lower VC dimension models
      => less hidden units(layers)
    - extract useful information during the model
      =>decreasing hidden units(neurons)
  - Regression:
    NN([in, 10, 5, 1],activations=['sigmoid', 'sigmoid', ‘linear'])
  - Classification:
    NN([in, 17, dim, 1],activations=['selu', ‘selu’, 'sigmoid'])

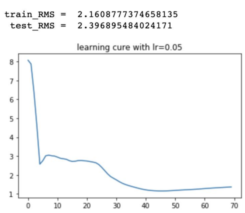
- Weight Initialization:
  - Xavier initialization / HE initialization

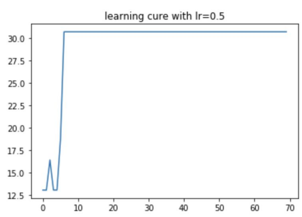
- ○ np.random.randn(layers[i+1], layers[i])*np.sqrt(2./layers[i]) for RELU-like activiations
- ○ np.random.randn(layers[i+1], layers[i])*np.sqrt(1./layers[i])) for Sigmoid-like activations
- ○ In the beginning, my model is quite sensitive to initial weights.
  And the model would stop working if bad init values are chosen.
  So I google for "weight initialization" and find the two heuristics depends on activation and the input size of the layer

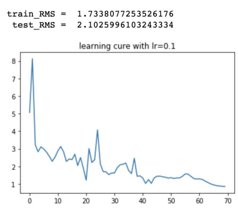
- Activation Functions:
  - ○ relu:
    zero gradient problem => try leaky relu/selu/sigmoid
    relu good enough for classification problem
  - ○ sigmoid:
    force the output to be in (0,1)
    not a good choice generally, but add a lot non-linearity in my opinion
    perform well in the regression task than relu/selu
  - ○ selu:
    A better version of relu-like functions, math backgrounded, self-normalizing.
    However, in my task, it seems that it just does similar job with RELU.

- Gradient Descent:
  Just the naive gradient descent with minibatch batch size = 10
  Nothing fancy.
  May add features of :
  - reuse data/emphasis data that didn't perform well,
    which is a similar idea to priority experience replay.
  - momentum/Adagrad/RMSprop/Adam...etc.

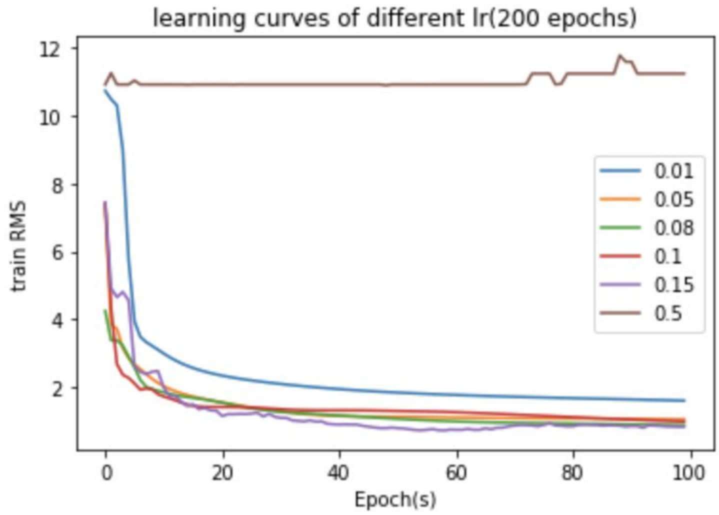
- Learning Rate:
  - ○ 0.5(too large): unstable and cause concussion and divergence
  - ○ 0.15(default): stable, generally good convergence results
  - ○ 0.1(default): stable, generally good convergence results
  - ○ 0.08(best): almost always good, smoother than 0.1, and slightly better RMS
  - ○ 0.01-0.05(too small): cause lower accuracy



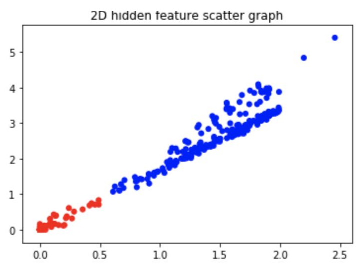


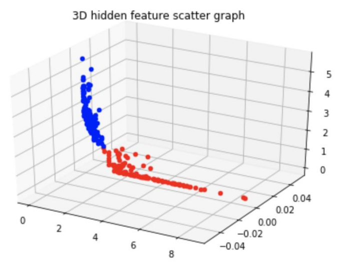


  - ○ In conclusion, 0.08-0.1 is good for both tasks in most cases



- Interesting facts about classification task:
  - it seems that the well-trained NN(because of better initialization) tends to form a line with a red-blue cut in 2D/3D space
    i.e. It is actually linear-separable!

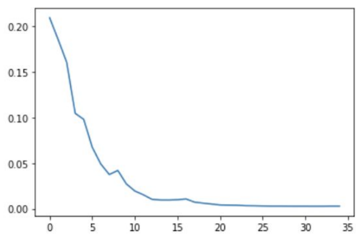




so I tried to set the last layer length to 1, and the result is good as I guess

```
In [79]: nn2 = NN([34, 17, 1, 1],activations=['selu', 'selu',  'sigmoid'], usage = 'classification')
         #the network architecture is as the constructer

         learning_curve = nn2.train(train_X, train_y, epochs=35, batch_size=10, lr = .1)

         train_CE = nn2.calc_error(train_X, train_y)
         train_accuracy = nn2.calc_accuracy(train_X, train_y)
         test_CE = nn2.calc_error(test_X, test_y)
         test_accuracy = nn2.calc_accuracy(test_X, test_y)

         plt.plot(np.arange(len(learning_curve)), learning_curve)

         print('train_CE = ', train_CE, '\n', 'test_RMS = ', test_CE)
         print('train_Accuracy = ', train_accuracy, '\n', 'test_Accuracy = ', test_accuracy)
         print('train_ErrorRate = ', 1-train_accuracy, '\n', 'test_ErrorRate = ', 1-test_accuracy)
```

```
train_CE =  0.005769448506493566
 test_RMS =  0.02004422382835881
train_Accuracy =  0.9857142857142858
 test_Accuracy =  0.8591549295774648
train_ErrorRate =  0.014285714285714235
 test_ErrorRate =  0.14084507042253525
```



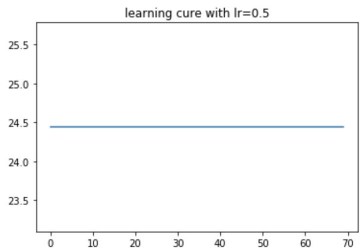
Moreover, as I remove the hidden layer (same as linear regression model with gradient descent) the accuracy is still quite high(92%/80%)
and one 3-node hidden layer can rise to (97%/85%)

In conclusion, I think the data is quite a trivial task: however, adding hidden layers can explore more inside properties and achieve a more precise result to determine “special cases” while preventing overfitting meanwhile.

Some Problems Encountered and Implementation Details:

- Numerical Issues:
    - exponential explosion
      => use numpy.clip(z, -500, 500) to avoid huge or close to zero values
    - division by zero(classification)
      => use numpy.clip(epsilon, 1-epsilon) to avoid 0 value of denominator
- Zero gradient:

```
train_RMS =  24.612000827283875
 test_RMS =  24.081340383946653
```



    - Wright initialization with HE/X => better result but still have zero gradient sometimes(on regression task)
    - SELU activation function => not working when bad init happens
    - use numpy.clip(z, -10, 10) to avoid zero gradient of exp related => not working
    - batch normalization layer is too complicated to implement
    - solved, bug with activation set to relu in the last layer

- @staticmethod decorator
- misunderstanding of np.linalg.norm(a-b) => correct RMS formula to np.linalg.norm(y-yhat)/np.sqrt(y.shape[1])
- np.where is useful :)
- save_res(name) function for Neural Network and prediction.csv saving
- Discover why zero gradient @ 2019.11.07 17:00.
  it is because I set the last-layer activation to relu instead of linear(None)
  this cause that about half probability the network will be “dead” due to initial weight’s output is negative
- There are still some numerical issues with selu if chosen as hidden activation in the regression model.

Other Issue:

- All work by myself with some Internet references, and TA consultation
- To run, need to move the corresponding data(.csv) and model(model.py) to the directory of ipython file
- For fast evaluation, cd reports