• Project title

Detection of mnist handwritten images

• Full name, student number.
Faezeh PouyaMehr 20241704
Kaggle username:
Faezeh

• Introduction: briefly describe the problem and summarize your approach and results.

In this project we design machine learning models that can accurately classify handwritten images of Mnist dataset which is a dataset of 50000 28*56 pixels grayscale images of handwritten two digits between 0 and 99 and the task is to classify a given image of handwritten digits into one of 19 classes which is the sum of digits in the image. In order to do the task after doing the EDA phase which contains the feature engineering and exploration of the data we perform three machine learning models namely multi-class logistic regression, cnn, and random forest to perform the classification and evaluate and analize accuracy and performance of each model. The results show that cnn models can outperform multiclass logistic regression and random forest model in this kind of problem. This is because its built-in convolutional layer reduces the high dimensionality of images without losing its information and divides the image into smaller areas in order to view them separately for the first time. That is why CNNs are especially suited for this use case and has the advantage to learn more complex, non-linear functions.

• Feature Design: Describe and justify your pre-processing methods, and how you designed and selected your features.

After observing few rows of data we first explore if the dataset contains any none entry which reveals that the last column is the case and we just remove it. After that we realize the data is already between 0 and 1 so we didn't do any normalization process. So at the end all the other features was selected for the learning process.

• Algorithms: give an overview of the learning algorithms used without going into too much detail, unless necessary to understand other details.

Multi class logistic regression

In statistics, multinomial logistic regression is a classification method that generalizes logistic regression to multiclass problems, i.e. with more than two possible discrete outcomes. That is, it is a model that is used to predict the probabilities of the different possible outcomes of a categorically distributed dependent variable, given a set of independent variables.

Step 1:

1. Insert a columns of ones to the first column of X to perform like a bias (w0) X: (number of samples, number of features +1)

2. Create a one hot encoding of our labels.
3. Initialize a weight matrix which store a vector in the dimension of our input X for each class k to update the weights in the gradient decent algorithm required to predict the class label for each given input test. The shape of Weights is (number of class, number of features+1)
4. Initiate alpha and lambda parameter which will be used for learning rate and regularization.

Step2:
For each class k we compute a linear combination of the input features and the weight vector of class k, that is, for each training example we compute a score that the input belongs to class k. For class k and input vector x(i) we have:

$\text{Prob\_val(k)} = W_k^T \cdot X^{(i)}$

Step3:
We apply softmax function to interpret this score into a probability value of a given input x belongs to class k. For class k and input vector x(i) we have:

$$P_k(x_i) = \frac{e^{\text{Prob\_val(k)}}}{\sum_{j=1}^{K} e^{\text{Prob\_val(j)}}}$$

Step4:

Using cross entropy loss over the whole training set we want the model to predict a high probability for the target class and a low probability for the other classes.

$$\text{Cost\_function(W)} = \frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{K} y_k^{(i)} \log(\text{Pk}(x_i)) + \frac{lam}{2*n} \|W\|^2$$

Step5:

Compute the gradient of the cost function with respect to each k weight vector.

$$\nabla_{w_k} cost\_function(W) = \frac{1}{n} \sum_{i=1}^{n} x^{(i)} * (\text{Pk}(x_i) - y_k^{(i)}) + \frac{lam}{n} W$$

Step6:

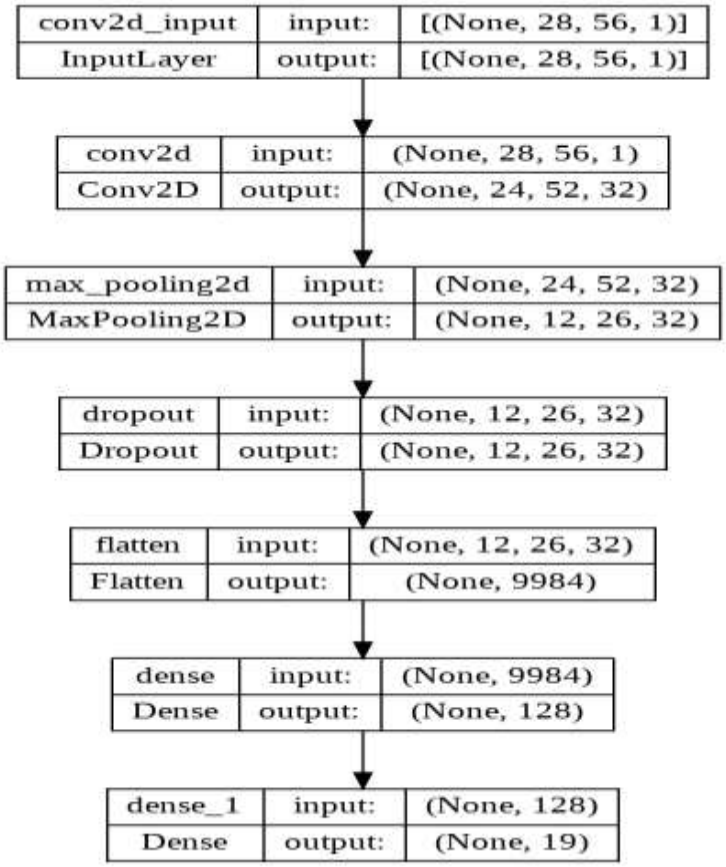
Updates the weights by using gradient decent algorithm:

$W_k = W_k - \text{alpha} * \nabla_{w_k} cost\_function(W)$

In addition to logistic regression we also implement cnn and RandomRorest models which were imported from defined library to be trained on the dataset.

CNN:

Our cnn model looks like this:



in which we use batch size of 128 and train on 300 epoch while using our validation_data=(valid_data, valid_label) by using categorical_crossentropy loss.

RandomForest:

Our randomForest classifer is imported form sklearn.ensemble getting n_estimators, random_state as its parameter to fit the training set.

• Methodology: include any decisions about training/validation split, regularization strategies, any optimization tricks, choice of hyper-parameters, etc.

we first shuffle the training point and then we split the 50000 samples into two sets of training and validation datasets. We use 80% of it for training and the remaining for the validation set using for tuning our hyper-parameters. We use regularization strategy for our logistic regression model in order to prevent overfitting. In addition we also use a initialization of our weight vector which is derived from a previously trained model the file's name is :"weight14s.npy"

Hyperparameter:

Logistic regression:
Alpha, lambda
We train the model on different values of both lam_value = [0.001,0.01,0.1] alpha_value = [0.01, 0.1, 0.25,0.5, 0.7] and evaluate the trained model on the validation set to find the best value for both parameters. The table1. is the validation accuracy obtained for each value of our hyperparameter. The loss function over iterations plot is shown in Fig7 to Fig 21 found in appendix for each value of our hyperparameter.

Lambda/alpha	0.01	0.1	0.25	0.5	0.7
0.001	0.1759	0.2164	0.2213	0.2059	0.1682
0.01	0.1759	0.2164	**0.2213**	0.2097	0.1839
0.1	0.1759	0.2164	0.2213	0.1968	0.185

Table1. Validation accuracy of logistic regression on different values for hyperparameters.

RandomForest:
n_estimators, random_state

we use values ranging from 400 to 600 with step size=15 as the n_estimators parameter and set the random_state=42+2*n_estimators.
The table2. is the validation accuracy obtained for each value of our hyperparameter:

N_estimator	**400**	**415**	**430**	**445**	**460**	**475**	**490**	**505**	**520**	**535**	**550**	**565**	**580**	**595**
Validation accuracy	0.74026666666666666	0.73906666666666666	0.74222	0.74113333333333333	0.73944	0.74206666666666667	0.74432	0.74413333333333333	0.74414	0.74373333333333334	0.74448	0.7441	0.74446	**0.74773333333333334**

• Results: present a detailed analysis of your results, including graphs and tables as appropriate. This analysis should be broader than just the Kaggle result: include a short comparison of the most important hyper-parameters and all methods (at least 2) you implemented.

Logistic regression:

The problem we faced in this project with logistic regression are due to its cons which are listed below:

- Cannot be applied on non-linear classification problems because it constructs linear decision boundry.
- Proper selection of features is required.
- Learning rate(α) and Regularization parameter(λ) have to be tuned properly to achieve high accuracy.
- It is tough to obtain complex relationships using logistic regression. More powerful and compact algorithms such as Neural Networks can easily outperform this algorithm
- Colinearity and outliers tampers the accuracy of LR model since this algorithm is sensitive to outliers
- Logistic Regression requires moderate or no multicollinearity between independent variables. This means if two independent variables have a high correlation, only one of them should be used. Repetition of information could lead to wrong training of parameters (weights) during minimizing the cost function.

One of the nice properties of logistic regression is that the logistic cost function (or max-entropy) is convex, and thus we are guaranteed to find the global cost minimum. But, once we stack logistic activation functions in a multi-layer neural network, we'll lose this convexity. However, the most important things to be considered in Logistic regression to achieve good accuracy is the proper selection of features.

Random Forest:

Random selection in individual decision trees of RFC can capture more complex feature patterns to provide the best accuracy. Random forests automatically create uncorrelated decision trees and carry out feature selection. It constructs each decision tree using a random set of features to accomplish this. This makes it a great model for working with data with many different features. RFC can also tell us how much each feature contributes to class prediction using Feature Importance and tree graphs for better interpretation which is why it achieves better accuracy that logistic regression.

Furthermore, Outliers do not significantly impact random forests. The variables are binned to achieve this. In addition, Both linear and non-linear relationships are well-handled by random forests.

Overall saying Random Forest Classifier performs better with more categorical data than numeric and logistic regression is a little confusing when comes to categorical data on the other hand Random forests perform better for a wide range of data items than a single decision tree. Both categorical and numerical data can be used with it. Furthermore, in most cases, no scaling or transformation of the variables is required.

However Random forest have some drawback which are listed below:

1. It requires much computational power as well as resources as it builds numerous trees to combine their outputs.
2. It also requires much time for training as it combines a lot of decision trees to determine the class.
3. Due to the ensemble of decision trees, it also suffers interpretability and fails to determine the significance of each variable.

CNN:

cnn models can outperform multiclass logistic regression and random forest model in this kind of problem. This is because its built-in convolutional layer reduces the high dimensionality of images without losing its information and divides the image into smaller areas in order to view them separately for the first time. That is why CNNs are especially suited for this use case and has the advantage to learn more complex, non-linear functions.

• Discussion: discuss the pros/cons of your approach and methodology and suggest ideas for improvement.

The cons of our logistic regression model is its slowness in convergence as it can be seen in its tiny improvement in learning training dataset in each iteration. The suggestion in this case is to use newton method instead of gradient decent algorithm. However, the runtime complexity of each iteration of newton method $O(nd(n+d)+d^3)$ (n is number of samples and d is the dimension of X)is bigger than the runtime complexity of each iteration of gradient decent O(nd), and also the memory complexity of each iteration in newton method $O(d^2)$ is bigger than in gradient decent O(d), which means newton method requires more operations and more amount of memory but it requires less number of iteration to converge to minimum because it uses the knowledge of the second derivative at each iteration while the gradient decent only uses the knowledge of the first derivative. This is all about the trade-off between the complexity and speed of convergence.

Another suggestion could be to select features wisely by considering the correlation of each pair of features or each feature with the label to only selects those who are relevant to the class labels and are not redundant. (Due to the high number of feature space I was not able to calculate this matrix because the google collab crashes)

For the cnn model to perfrom better we could add another conv layer or increase the number of iteration.

• Statement of Contributions. add the following statement: “I hereby state that all the work presented in this report is that of the author”.
I hereby state that all the work presented in this report is that of the author

• References: very important if you use ideas and methods that you found in some paper or online; it is a matter of academic integrity.
https://medium.com/@papillonbee/logistic-regression-from-scratch-with-gradient-descent-and-newtons-method-ff4307e3cb30

https://www.kaggle.com/code/blurredmachine/mnist-classification-eda-pca-cnn-99-7-score

https://subscription.packtpub.com/book/big-data-&-business-intelligence/9781788397872/1/ch01lvl1sec27/pros-and-cons-of-neural-networks

https://github.com/bamtak/machine-learning-implemetation-python

• Appendix (optional).

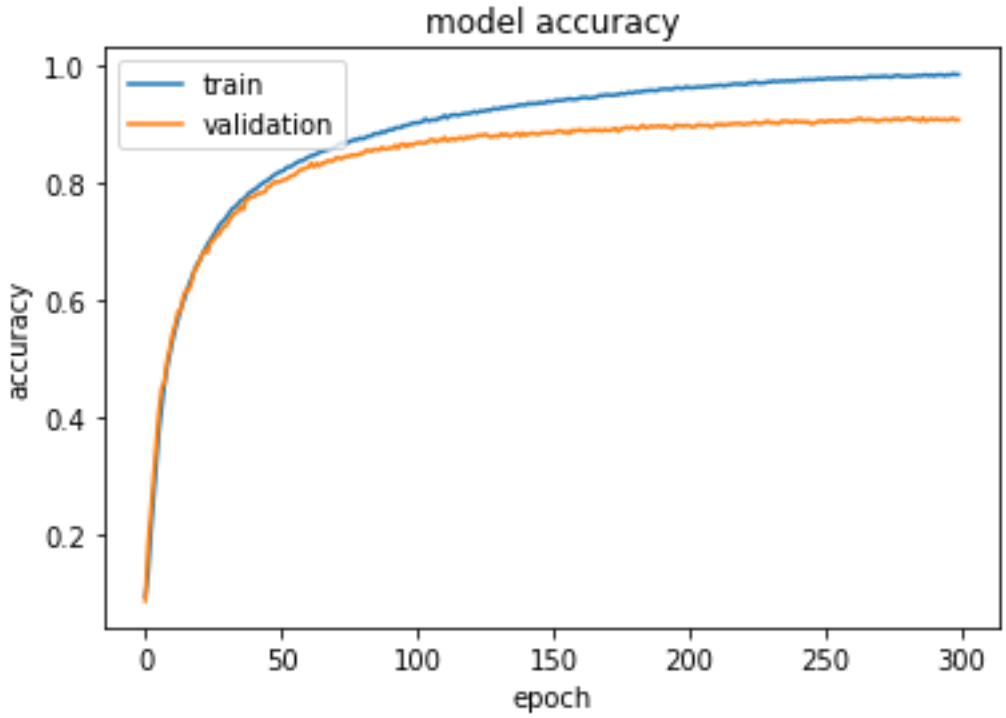


Figure 1.cnn model accuracy over iterations

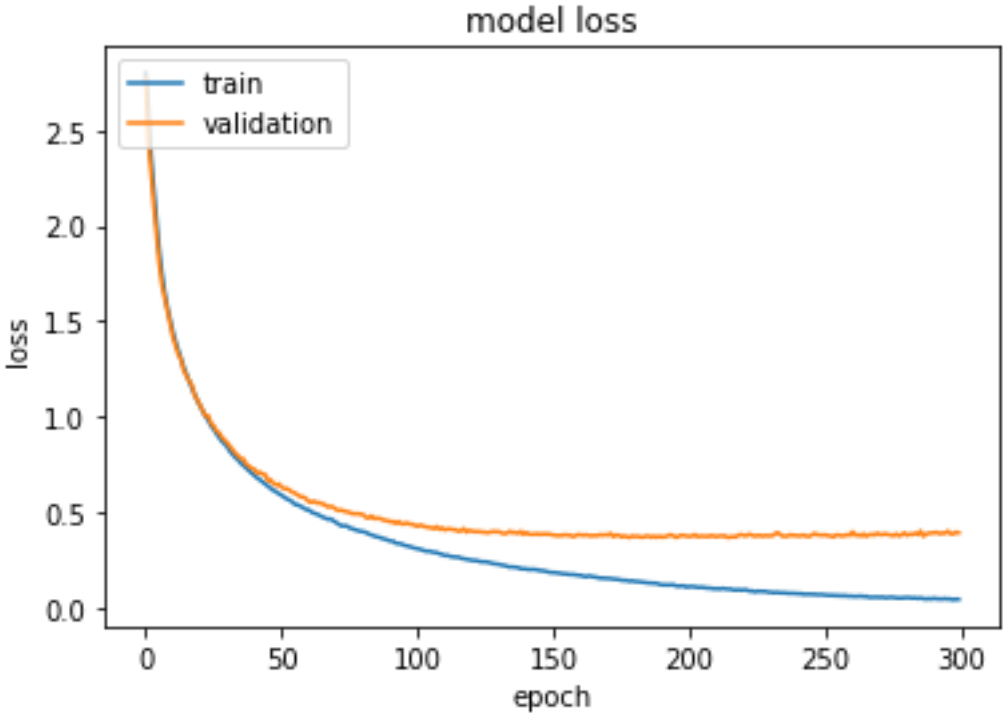


Figure 2.cnn model loss over iterations

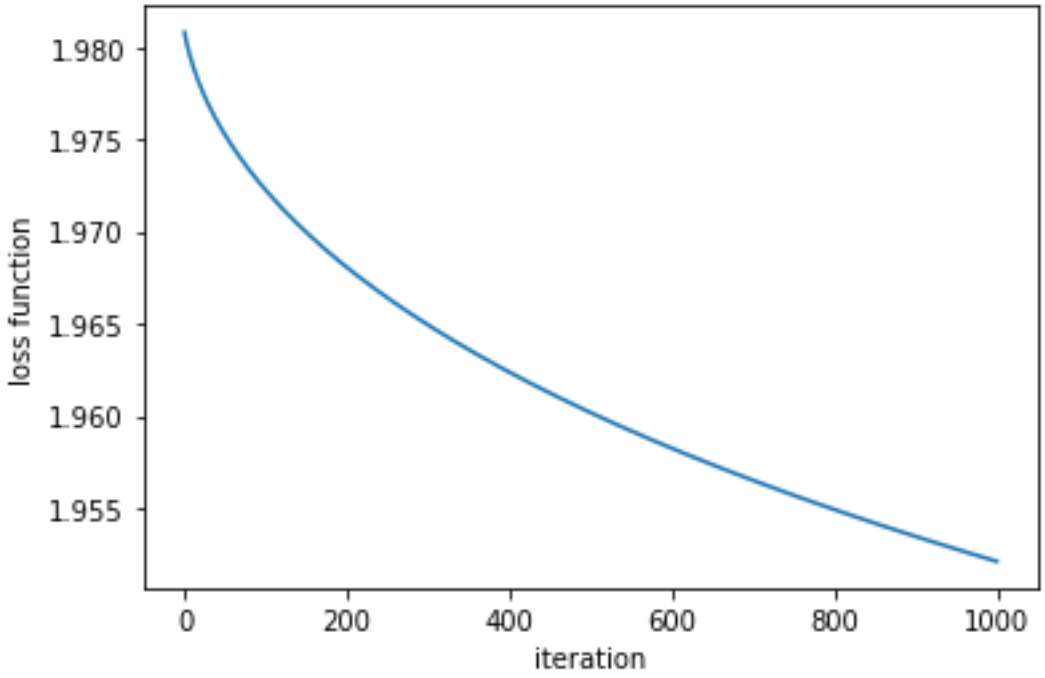


Figure 3.logistic reg loss_train_on_star_params

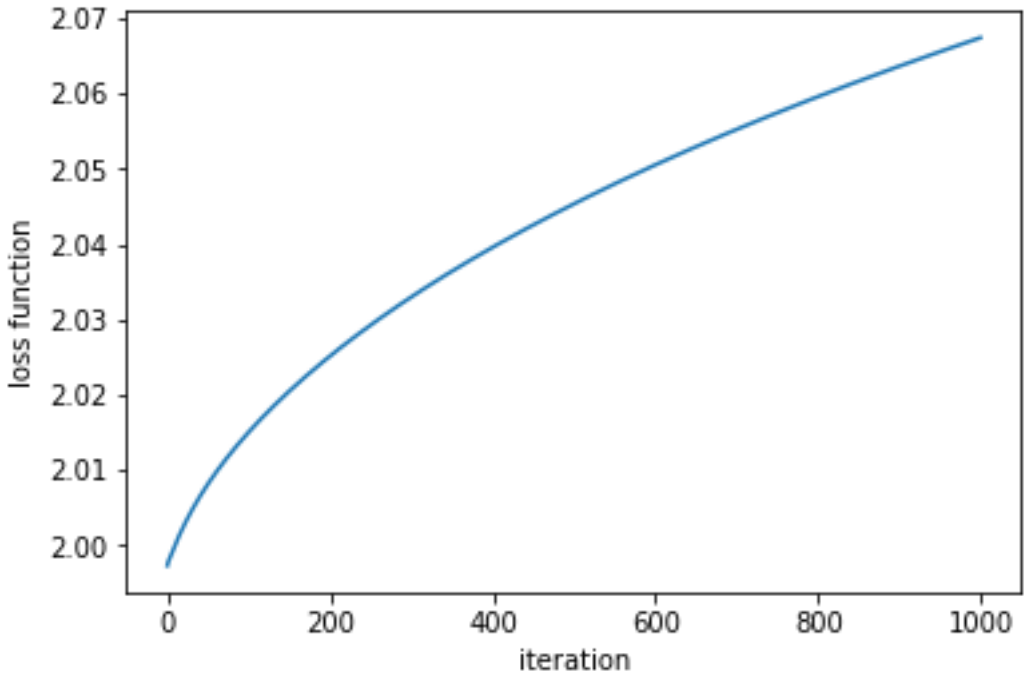


Figure 4.logistic reg loss_valid_on_star_params

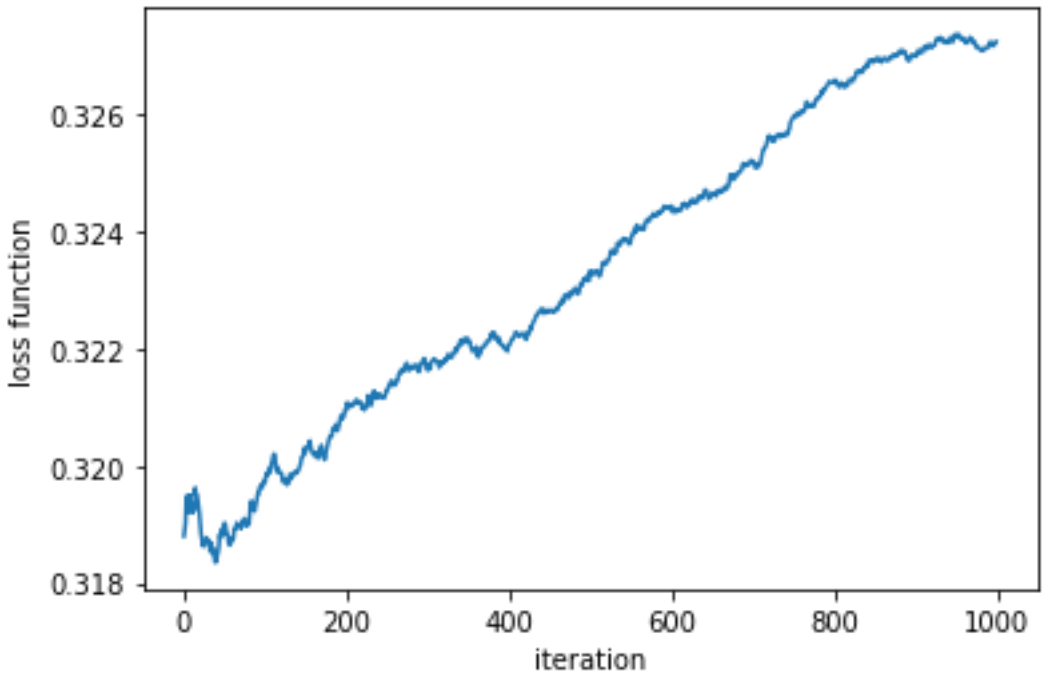


Figure 5.logistic reg acc_train_on_star_params

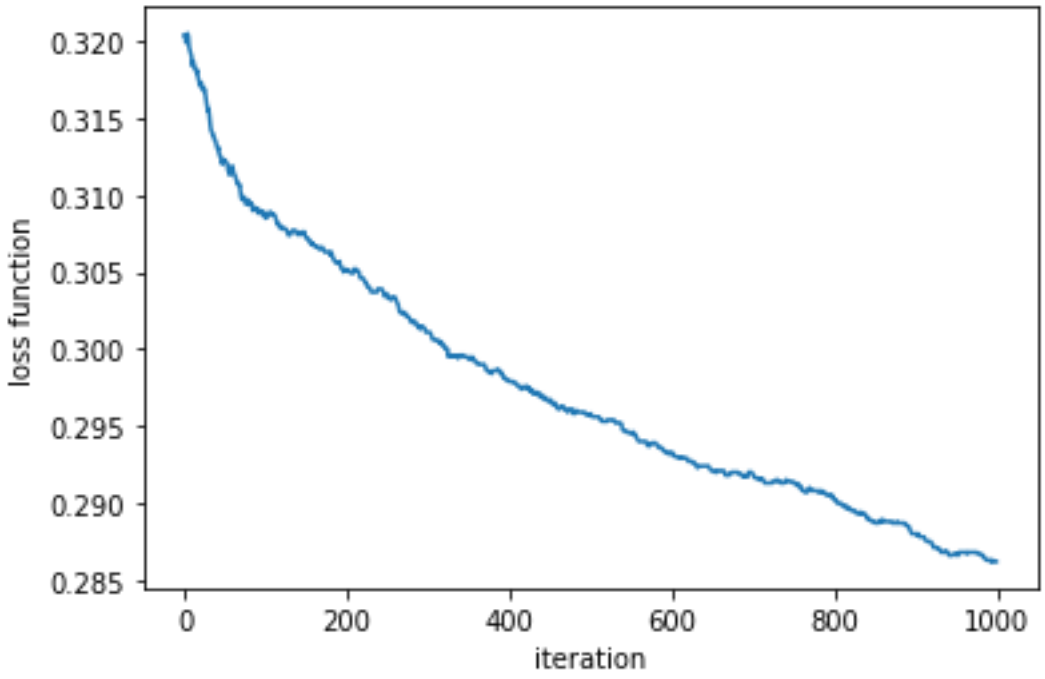


Figure 6.logistic reg acc_valid_on_star_params

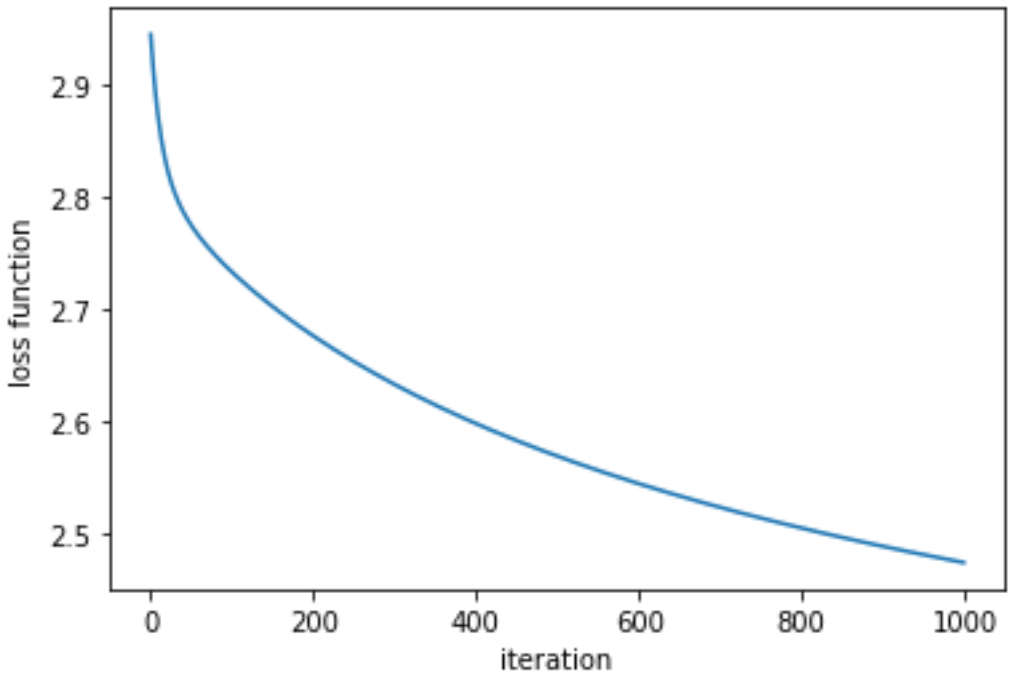


Figure 7.hyper parameter tuning loss function with .lambda=0.001 alpha=0.01

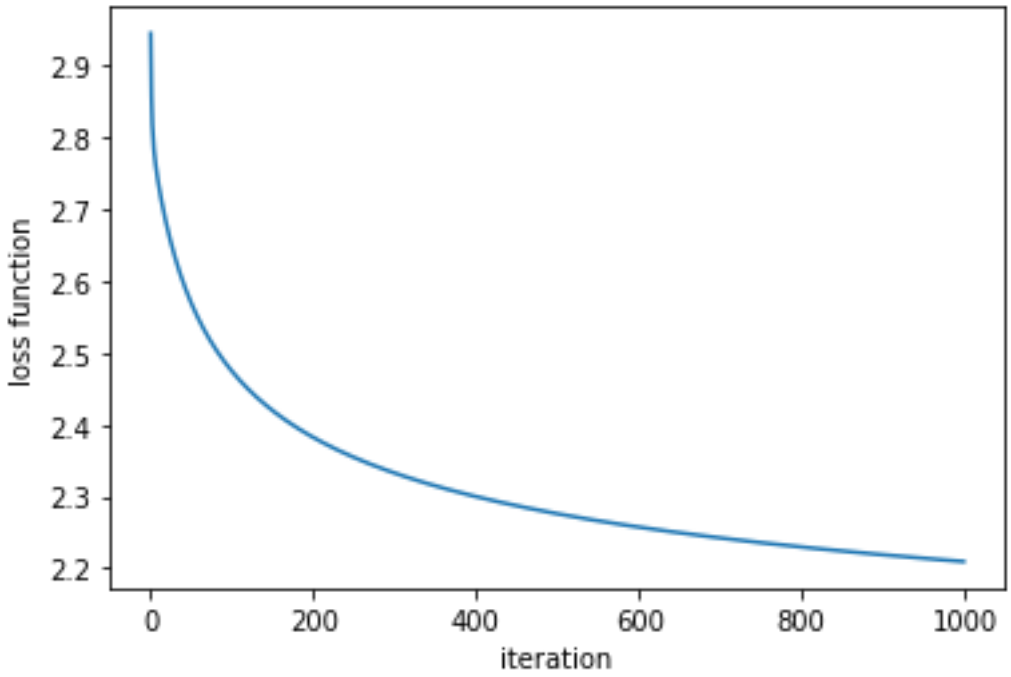


Figure 8. hyper parameter tuning loss function with lambda 0.001_alpha_0.1

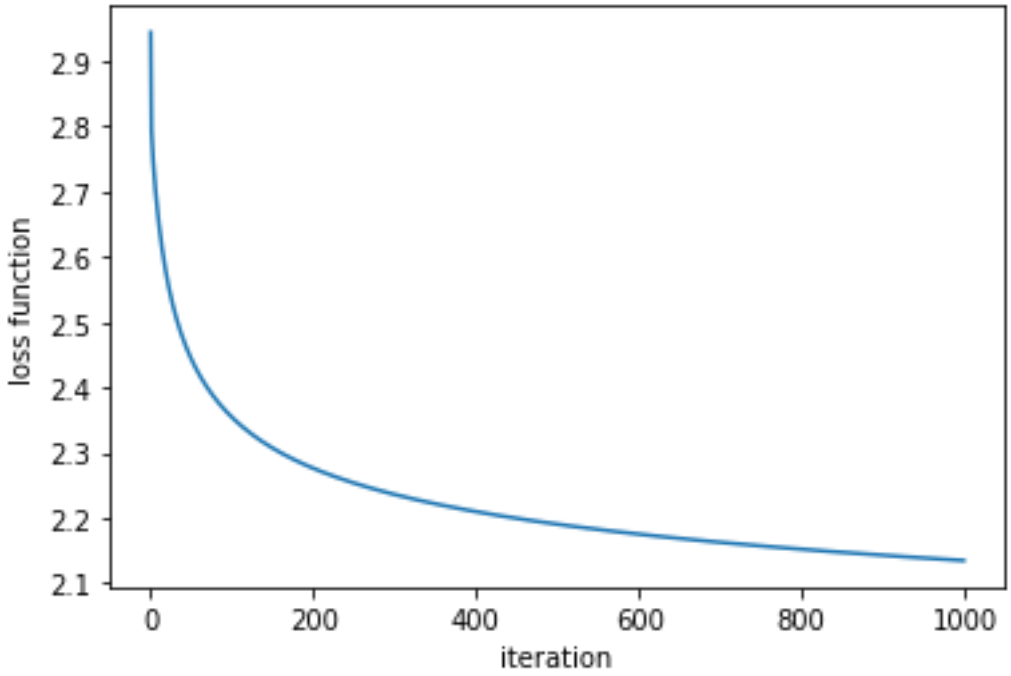


Figure 9. hyper parameter tuning loss function with lambda_0.001_alpha_0.25

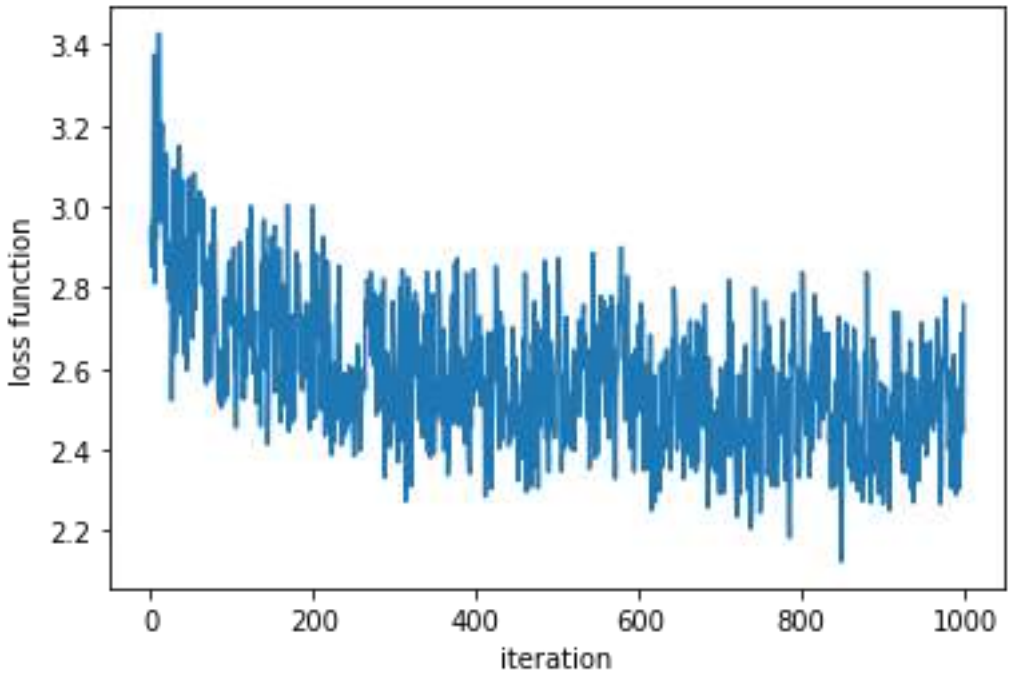


Figure 10. hyper parameter tuning loss function with lambda_0.001_alpha_0.5

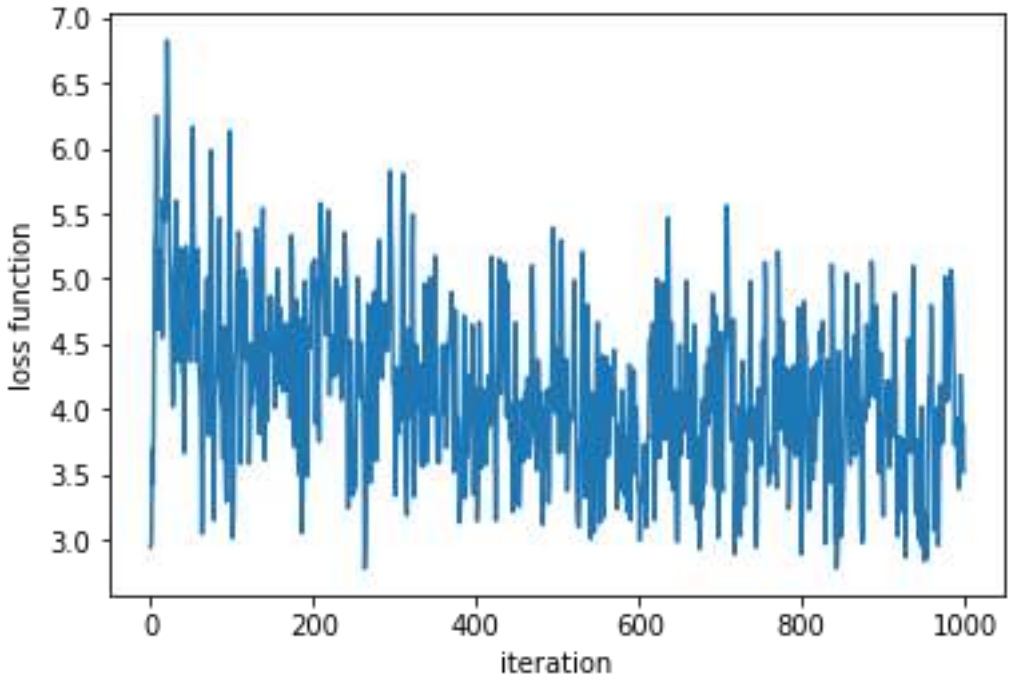


Figure 11. hyper parameter tuning loss function with lambda_0.001_alpha_0.7

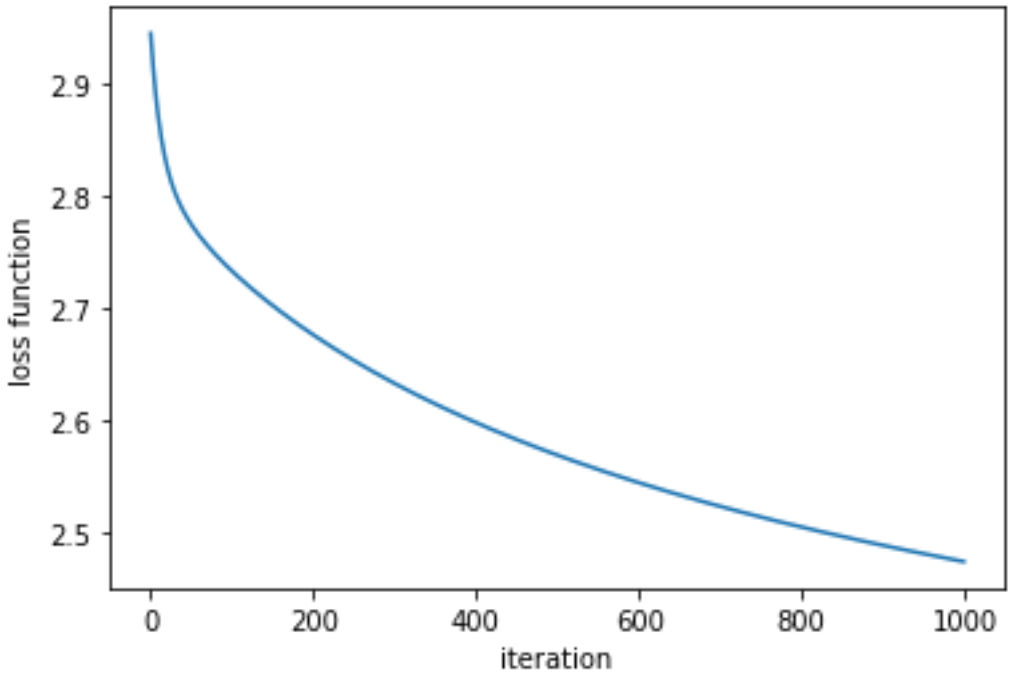


Figure 12. hyper parameter tuning loss function with lambda_0.01_alpha_0.01

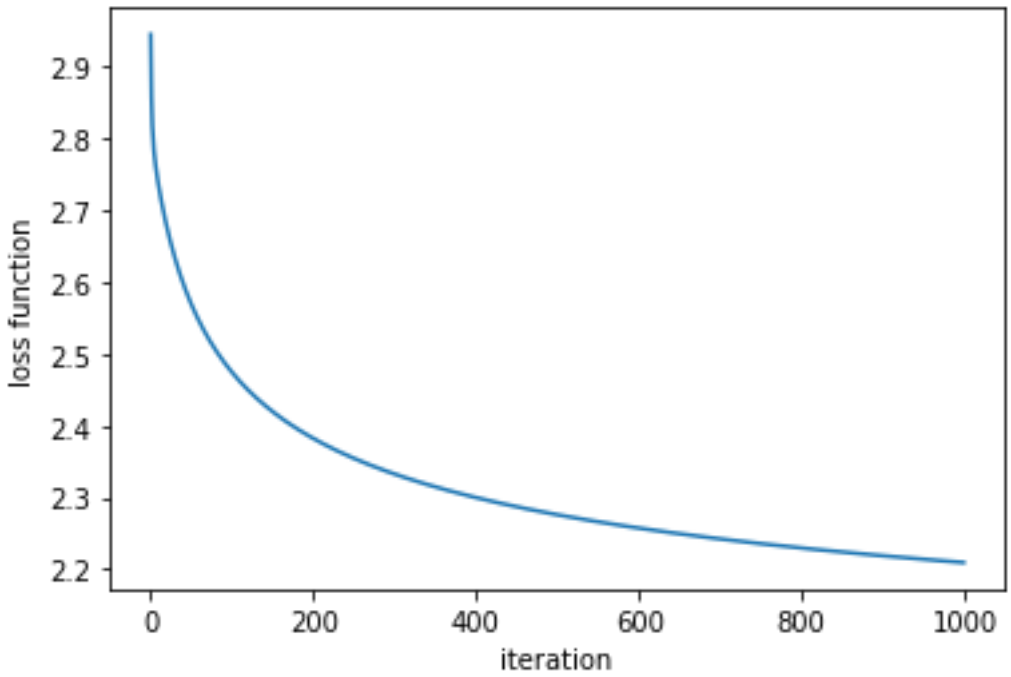


Figure 13. hyper parameter tuning loss function with lambda_0.01_alpha_0.1

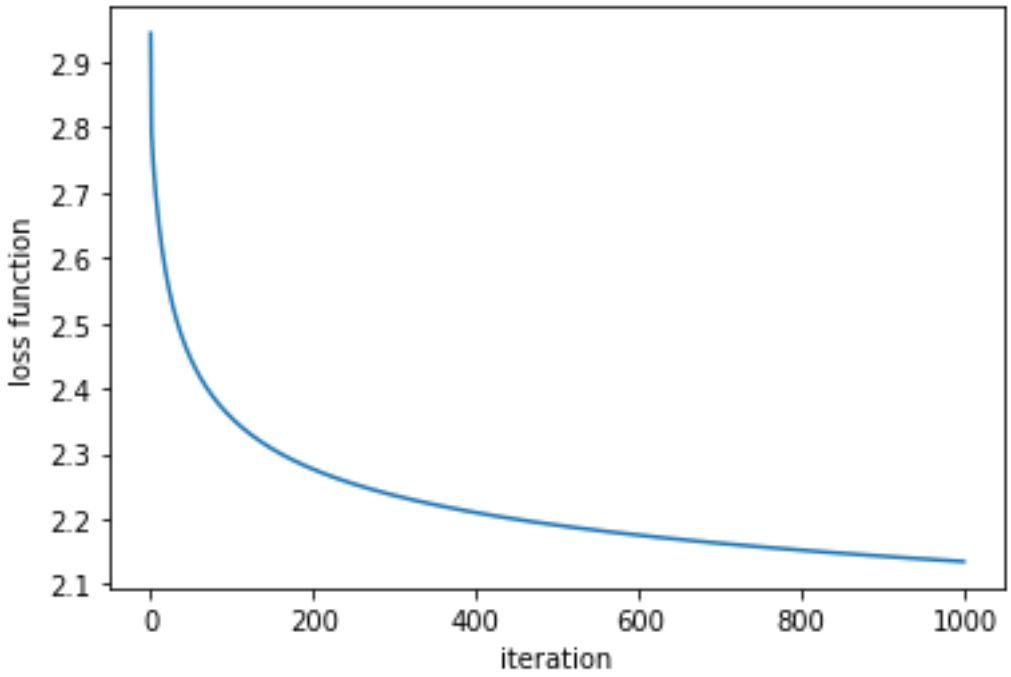


Figure 14. hyper parameter tuning loss function with lambda_0.01_alpha_0.25

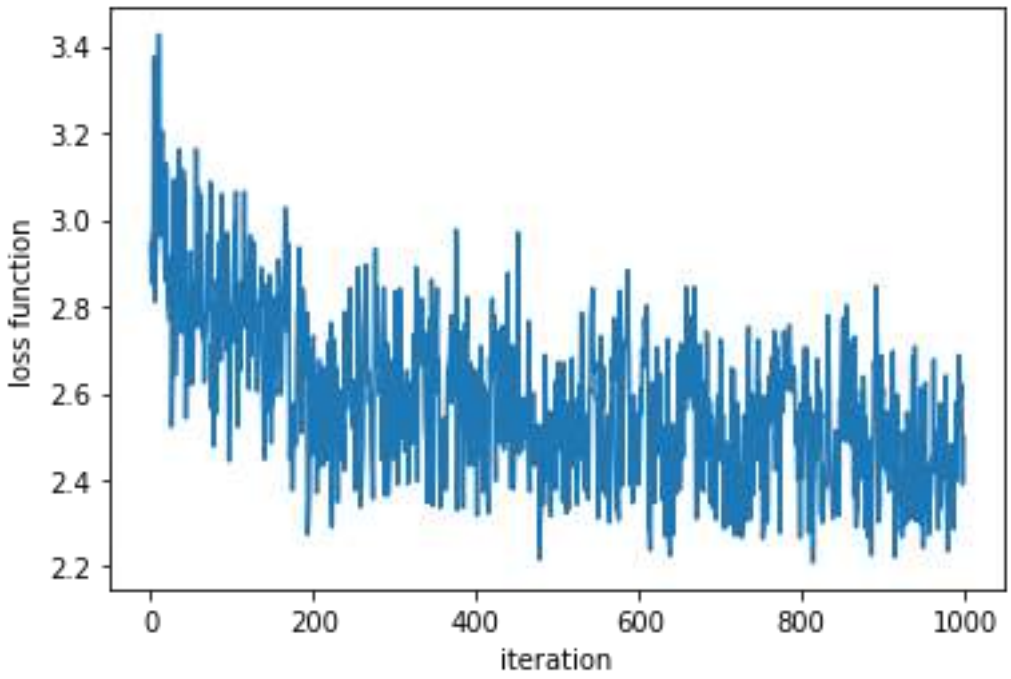


Figure 15. hyper parameter tuning loss function with lambda_0.01_alpha_0.5

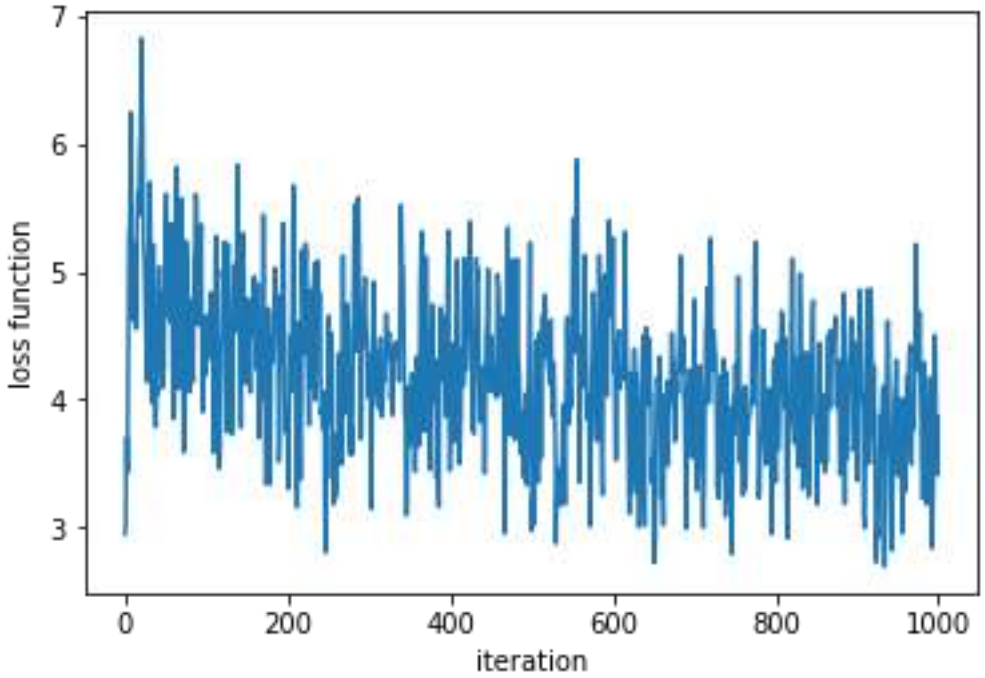


Figure 16. hyper parameter tuning loss function with lambda_0.01_alpha_0.7

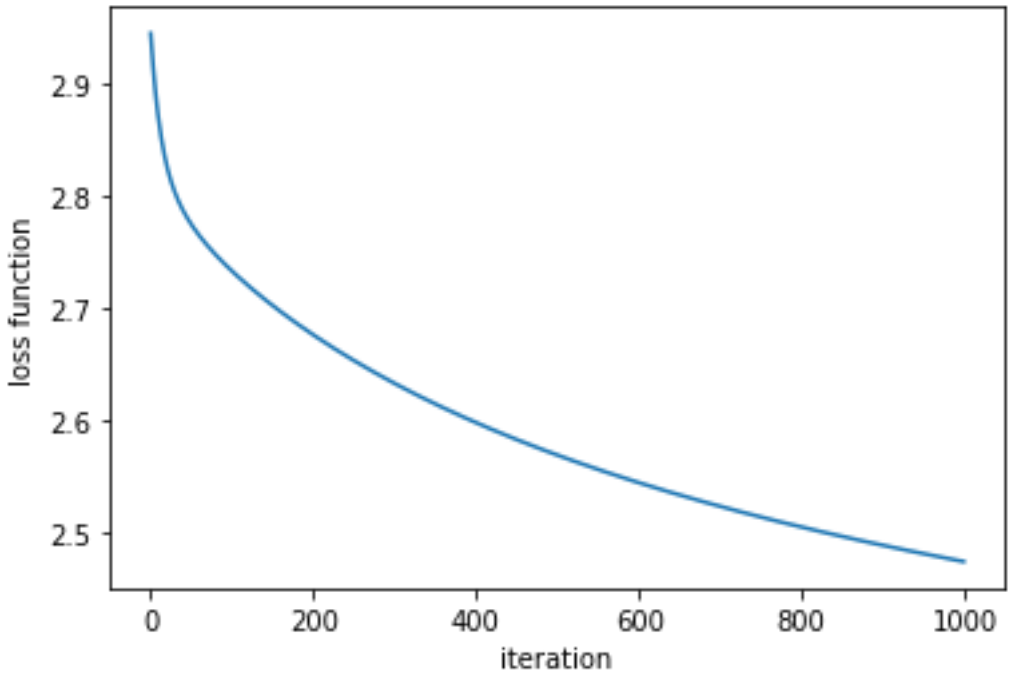


Figure 17. hyper parameter tuning loss function with lambda_0.1_alpha_0.01

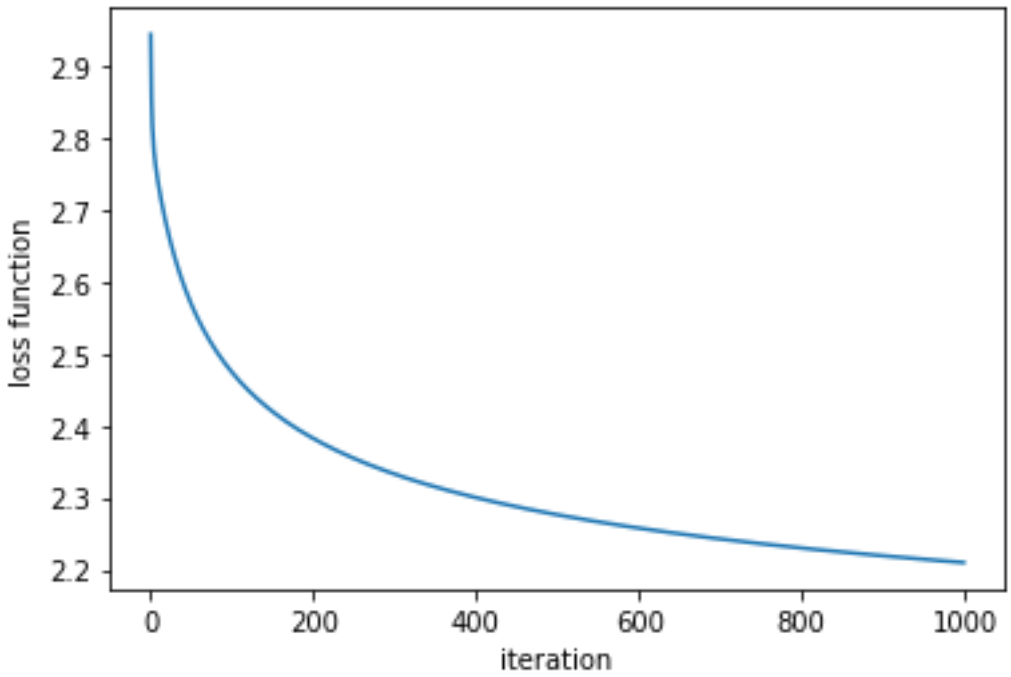


Figure 18. hyper parameter tuning loss function with lambda_0.1_alpha_0.1

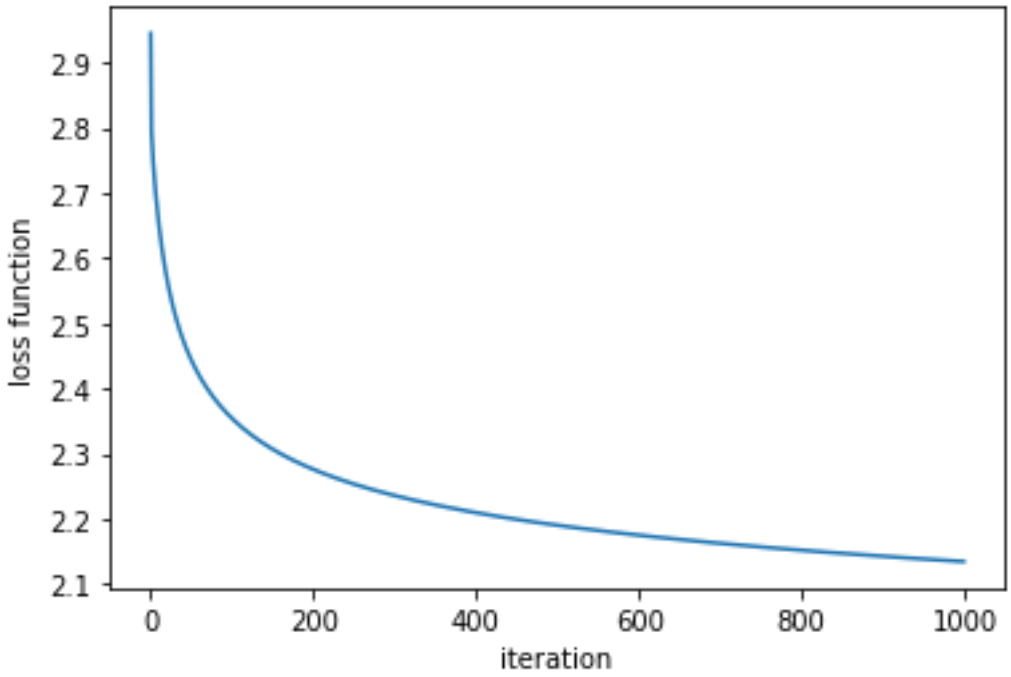


Figure 19. hyper parameter tuning loss function with lambda_0.1_alpha_0.25

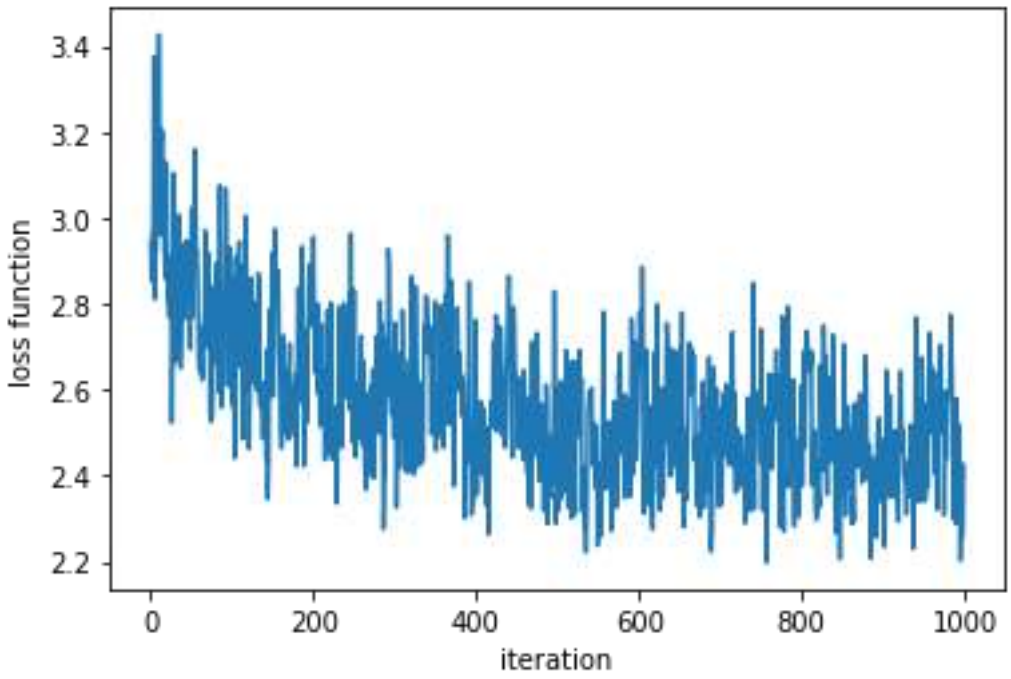


Figure 20. hyper parameter tuning loss function with lambda_0.1_alpha_0.5

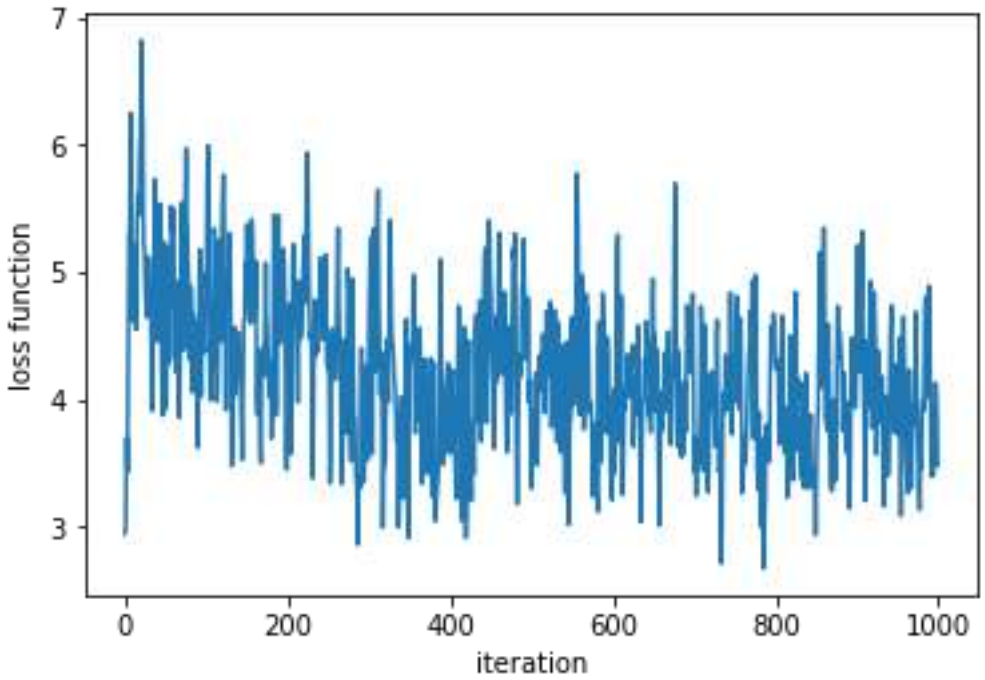


Figure 21. hyper parameter tuning loss function with lambda_0.1_alpha_0.7