CS577: Assignment 3

Jane Downer
Department of Computer Science
Illinois Institute of Technology

October 25, 2022
(used 2 late days)

Abstract
Collect data for both classification and regression tasks. Observe the effects of incorporating different loss functions, optimizers, and regularization strategies into the models.

1 Problem Statement

- Find the combination of loss, optimization, and regularization parameters that optimize model performance. Repeat for both classification and regression data.

2 Proposed Solution

- Build basic models depending on which parameters we are testing. Write functions so model structures can be reused with different values if needed. Iteratively modify the original model with a grid search and keep track of performance. Plot performance of different models against each other.

3 Implementation Details

Before running code, set 'ROOT_PATH' at top of file to your preferred local directory.

1. Import data
   - Classification data: UCI's Iris dataset [1]
     - 150 observations, 4 attributes, 3 categories
   - Regression data: UCI's Community dataset [2]
     - 1994 observations, 122 attributes (128 originally, but we only use numeric attributes)
   - Data prep:
     - Shuffle dataset
     - Convert numerical data to float (in case not already correct type)
     - Split data into train and test sets
     - Classification data only:
       - Perform one-hot encoding on labels
     - Regression data only:
       - Replace any '?' entries with np.NAN
       - Use sklearn.imput.SimpleImputer to replace np.NAN values with column averages
2. Loss: train for various combinations of loss functions and hyperparameters
   - Build Model:
     - Dense layer + ReLU
     - Dense layer + ReLU
     - Dense layer + Output Activation*
   - Perform grid search on the following:

• Categorical loss functions:	Categorical Cross-Entropy, Categorical Hinge, KL-Divergence
• Regression loss functions:	Log-cosh, MAE, MSE
• Learning rates:	0.1, 0.01. 0.001
• Batch sizes:	16, 32, 64
• Hidden units:	(32,16), (64,32)

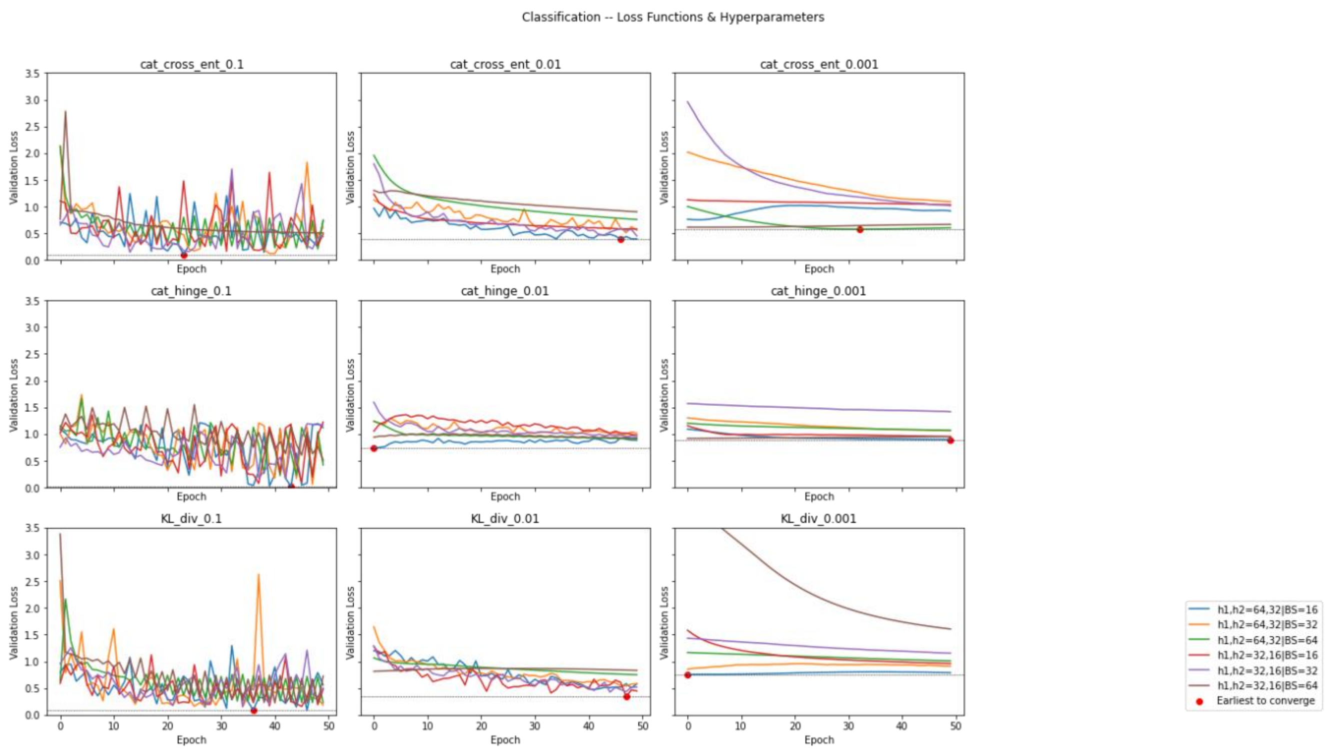
   - Optimizer: SGD for now
   - Regularization: None for now

3. Optimization: using best loss function/hyperparameter combination(s), try various optimizers.
   - Build Model:

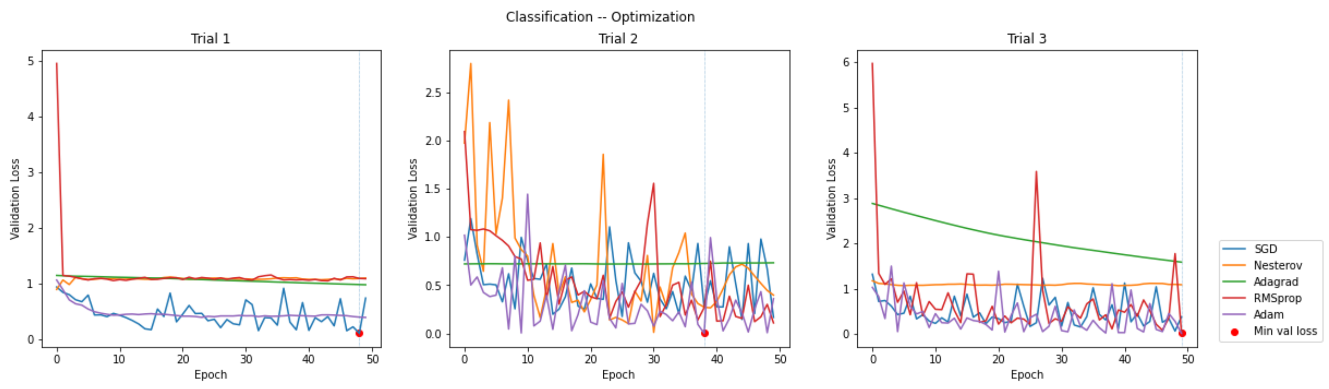
        - Same model structure as before, but compile on different optimizers.
    - Optimizers: SGD, Nesterov, Adagrad, RMSprop, Adam
    - For each optimizer, find epoch and time where loss is minimized
    - Results vary because of randomness – run 3 trials and choose optimizer that is most frequently best
4. Regularization: using best loss function/hyperparameter/optimizer combination(s), try different regularization methods.
    - 6 models:
        - No regularization
        - Batch Normalization
        - Dropout (p=0.5)
        - Weight Decay (decay = 1e-06)
        - Batch Normalization + Dropout + Weight Decay
        - Ensemble model of the 5 models above
            - New training labels = average of model outputs
            - Build, compile and fit new model with same training data as X_train and new training labels as y_train

4 Results and Discussion

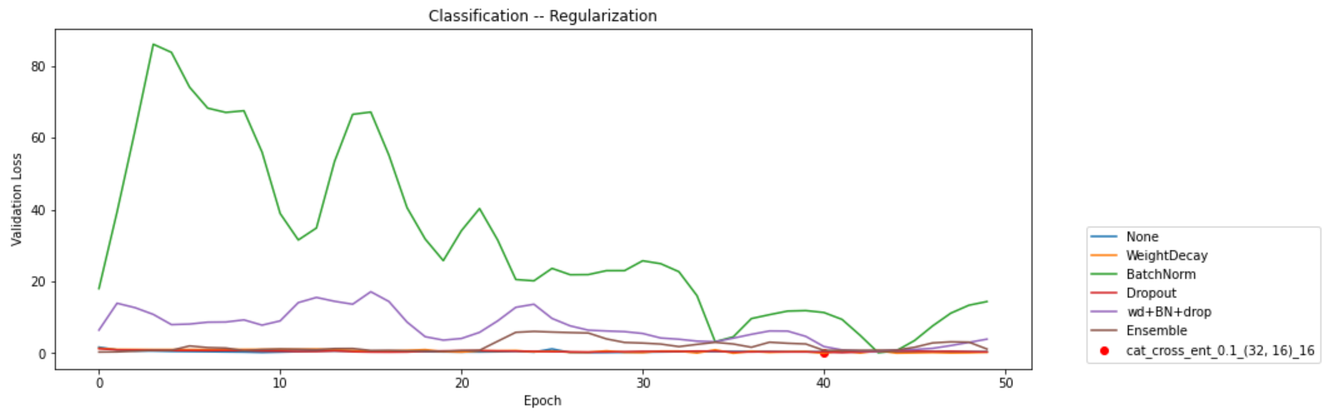
Classification



Here are the graphs of each combination of loss, learning rate, number of hidden units, and batch size. I graphed the data by loss function and learning rate to make the data more legible. Larger learning rates resulted in faster convergence but smaller learning rates resulted in smoother trends. Categorical hinge loss reached the lowest minimum loss value, but categorical cross entropy reached a similarly-low value in in fewer epochs (23 epochs versus 43) and a shorter time (1.317 seconds versus 1.937 seconds). I will use Categorical cross entropy moving forward for this reason. The model performed better with fewer hidden units (32 and 16 rather than 64 and 32 and smaller batch sizes (16 rather than 32 or 64). This makes sense because the dataset is small.



Testing different optimizers produced very different results each time due to randomness, so I ran three trials and looked at overall trends. RMSprop, Adagrad, and Nesterov performed poorly in compared to the other optimizers. In each of the 3 trials, Adam resulted in the lowest validation loss in 2 of 3 trials – this makes sense, because it combines the benefits provide by the other optimizers.



There is some randomness associated with these regularization functions, and the results look a bit different each time. Using batch normalization results in a lot of fluctuation over the 50 epochs. Using a combination of weight decay, dropout, and batch normalization also results some fluctuation, but not as much. I Googled the issue with batch normalization, and found some research [3] saying it might be problematic for small datasets – this dataset has only 150 examples, so perhaps that is the reason batch normalization isn't helping.

The absolute minimum loss value is reached using weight decay. However, using dropout, weight decay, and no regularization method at all reach similar validation loss values almost immediately.

Classification Model Evaluation

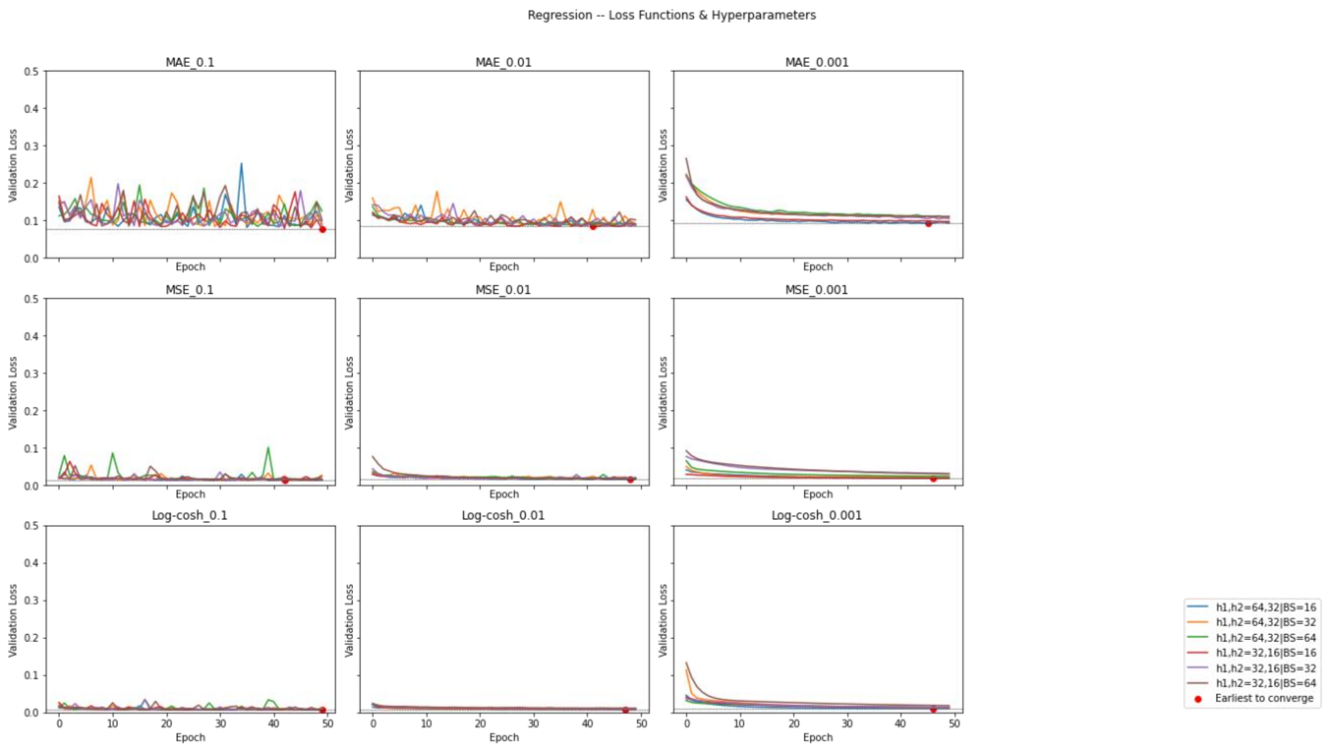
Calling the following line of code:

```
best_clf_model.evaluate(X_tst_iris, y_tst_iris)
```

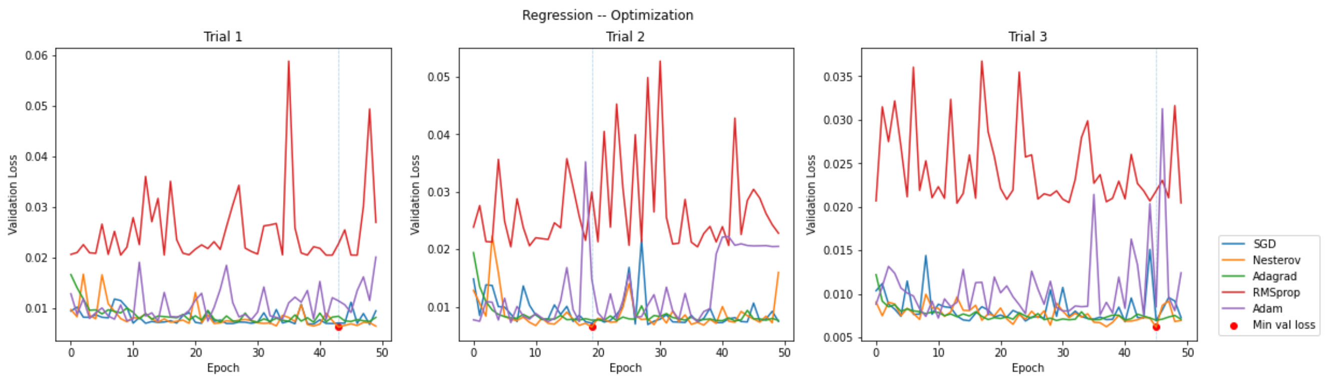
Tells us that the model reaches 97% accuracy on the classification test set:

```
4/4 [==============================] - 0s 3ms/step - loss: 0.0683 - accuracy: 0.9700
[0.0683361068367958, 0.9700000286102295]
```

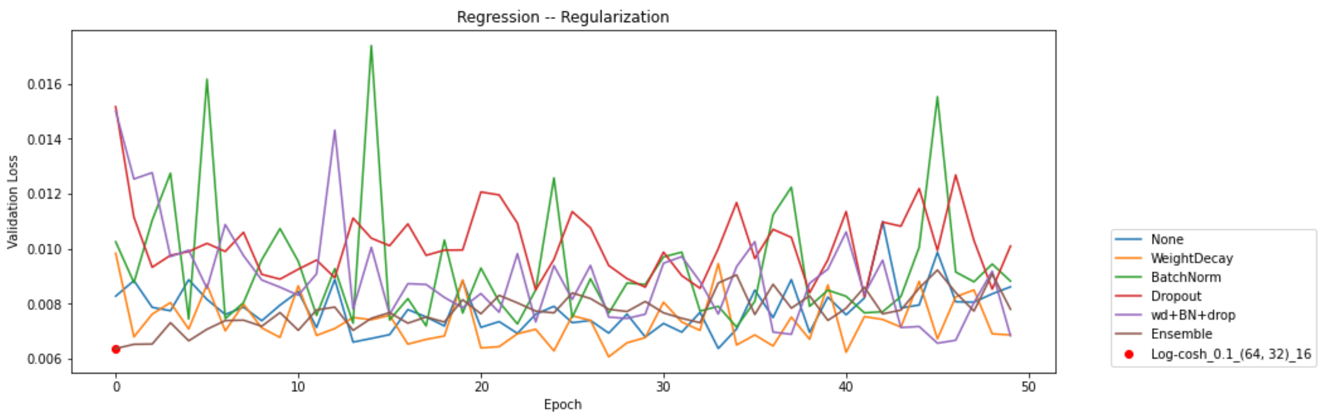
Regression



For the regression dataset, log-cosh performs best, followed by MSE and MAE. The lowest validation loss is achieved when the learning rate is 0.1, number of units in hidden layer are 64 and 32, and batch size is 16.



As with the classification data, I did 3 trials for the regression optimization task. RMSprop performs consistently worse than the others. With the other optimization methods, validation loss reaches low values quickly (between 10 and 20 epochs) in each trial. Nesterov reached the lowest validation loss in all 3 cases. Since Adam uses some of the features from RMSprop, and RMSprop performs poorly here, it makes sense that Adam's performance is noisy. However, the reason RMSprop is performing poorly is unclear.



This graph shows the result of using various regularization methods while training – the process is the same as it was for the classification task. The smallest validation loss is reached with the ensemble method at the first epoch. Interestingly, the ensemble method (along with the model using no regularization at all) seems to have a slight upward trend This is what we expect, because it combines the benefit of the other regularization methods.

There were a significant number of missing values in the original dataset, which I replaced with the column averages. I wonder if this interpolation helped bring stability to model performance.

<u>Regression Model Evaluation</u>

We can define the model that combines all of the above features (for regression) as `best_reg_model`. Using the following formula for accuracy:

```
y_preds = best_reg_model(X_tst_community)
y_preds = [0 if row[0]<0.5 else 1 for row in y_preds]
y_tst_binary = [0 if row[0]<0.5 else 1 for row in y_tst_community]
acc = np.average([1 if y_preds[i]==y_tst_binary[i] else 0 for i in range(len(y_preds))])
```

We find that the model yields 96.6% accuracy on the regression test set. (See notebook.)

```
Accuracy: 0.966
```

5 Conclusion

For the classification task, the following configuration led to the best performance on the validation set:

- Loss: Categorical Cross Entropy
- Optimizer: Adam
- Regularization: weight decay, dropout, none
- Learning rate: 0.1
- Batch size: 16
- Units in first and second hidden layer: 32,16

For the regression task, the following configuration led to the best performance on the validation set:

- Loss: log-cosh
- Optimizer: Nesterov
- Regularization: ensemble
- Learning rate: 0.1
- Batch size: 16
- Units in first and second hidden layer: 64, 32

6 References

[1] https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data

[2] https://archive.ics.uci.edu/ml/machine-learning-databases/communities/communities.data

[3] http://proceedings.mlr.press/v89/lian19a/lian19a.pdf