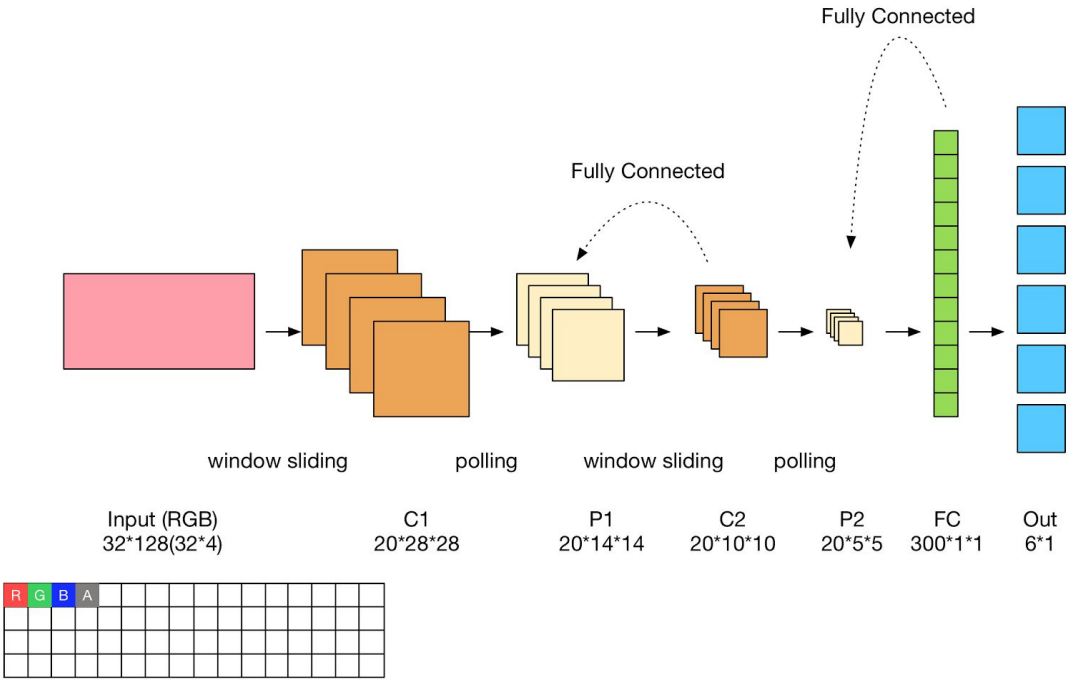
Lab3 Report

Group members: Feng Jiang, Zijie(Jay) Wang

Topology



Basically we followed the same topology shown in the slides in the lecture. Detailed size and design are drawn above. We chose to use the C-P-C-P-F format, since professor said it was a more “common” format in the field.

For the Input layer, we used a 2D array to represent the raw feature vector. Since we were using the RGB-Gray color, the 2D array is not a square but a rectangle whose length was four times (four entries in one pixel) longer than the image length. In the actual implementation, we use a special stride to do the window sliding. For convolutional and pooling layers, and the fully connected layer, we chose the same size and plate numbers as shown on the slides. We had 6 categories to predict in this project, so we have 6 units in the output layer with one-hot encoding.

Implementation Notes

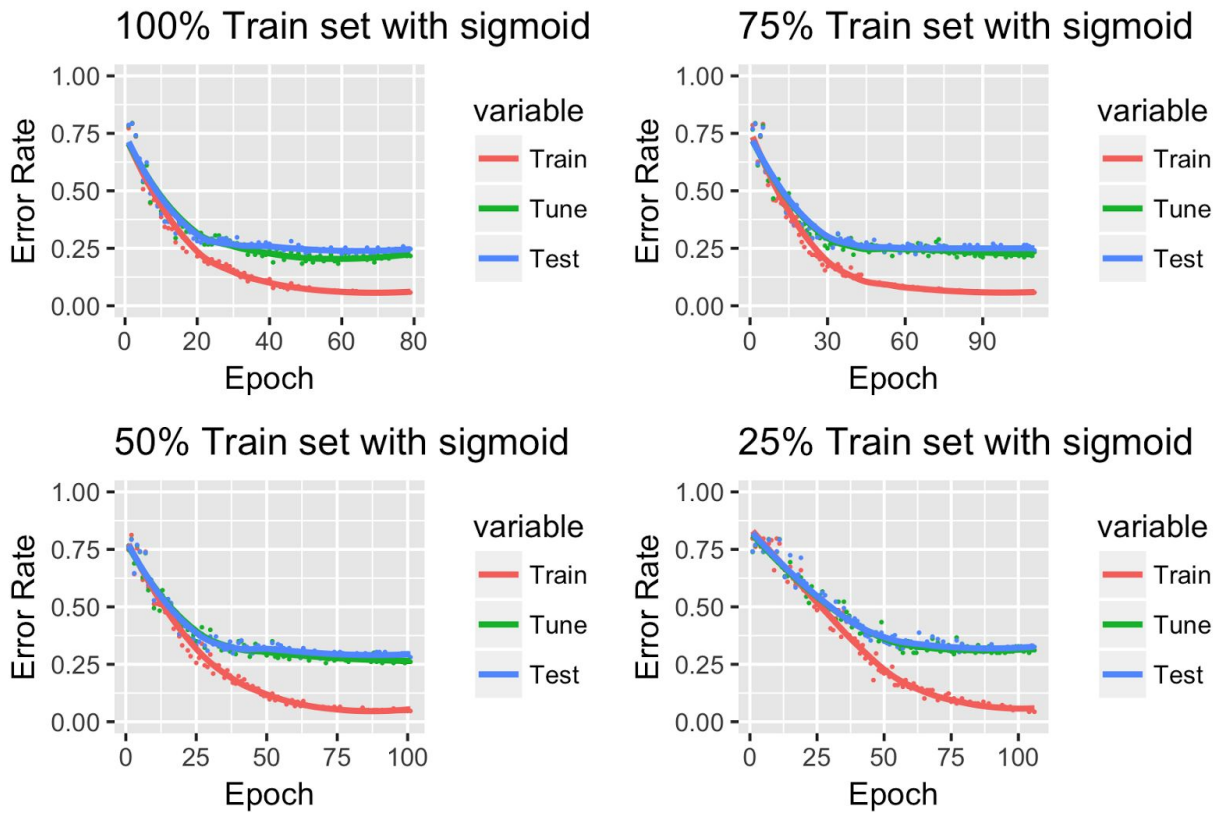
This project was very challenging for us, and we have spent 3 weeks and more than 50 hours on this project and while wrote three versions of it. But finally we had great results. We list some interesting findings as below:

1. If we chose to use Leaky Rectifier or Rectifier as our activation function, then we must chose a very small learning rate. Otherwise the weights would overshoot to infinity in the early stage. This was hard to debug, since the error we got was "ArrayIndexOutOfBound". For the same reason, it was better to use sigmoid function initially.
2. It was very easy to make mistakes when matching the index of each matrices and for loop. One way to cope with that was to fully understand the details of how each layer works before writing code.
3. Infinite Gradient Checking was actually hard to use. For different activation function, it required different loss functions, which sometimes could be extremely confusing.
4. Extra input example was extremely helpful, which bumped up our accuracy by about 3~4%.

Learning Curve

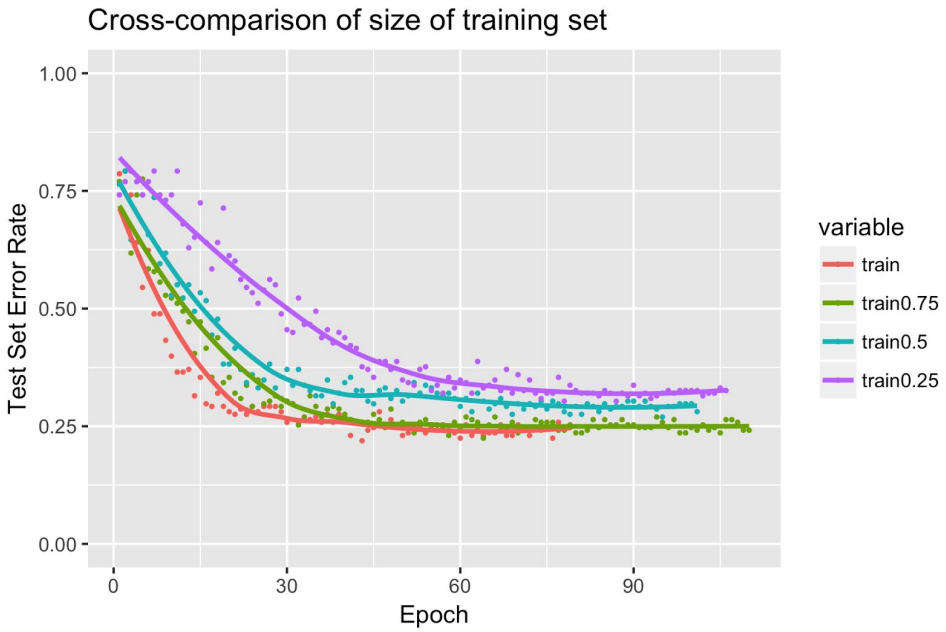
We have done some experiments to see how the size of training examples affect the learning process.

1. **Learning curves of using different proportions of the training set**



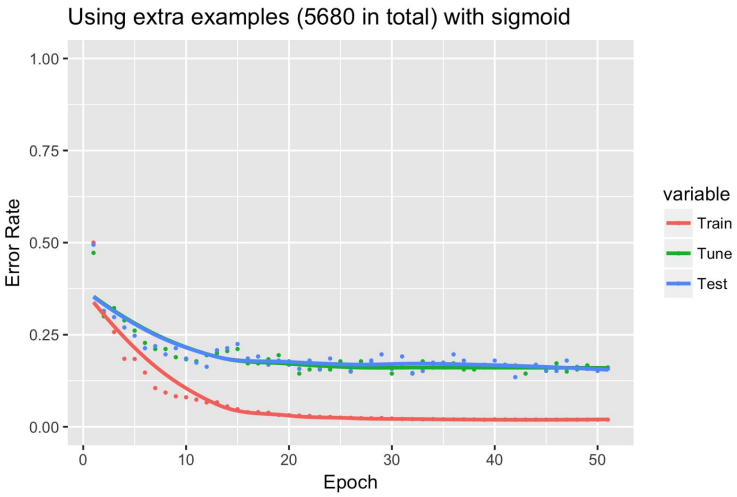
The error rate did not really increased and overfitting happened earlier (the differences of error rates between training set and testing set) when we used lesser examples. It was slower for CNN to converge while using a subset of training samples and the needed epoch increases. Compared to using 100% training examples, error rates have a higher variance when using 75% and 25% training examples.

2. **Cross comparison of test set error rates with different proportions of the training set**



As we expected, the test set error rates increased when we used less examples. It also took longer time to converge to the final best plateau when we used less examples. The test set error rate, and the epoch for achieving the best tune set error rate for different portions of examples (100%, 75%, 50%, 25%) were 23.59% at 49 epoch, 25.28% at 90 epoch, 26.96% at 95 epoch, and 32.02% at 76 epoch respectively.

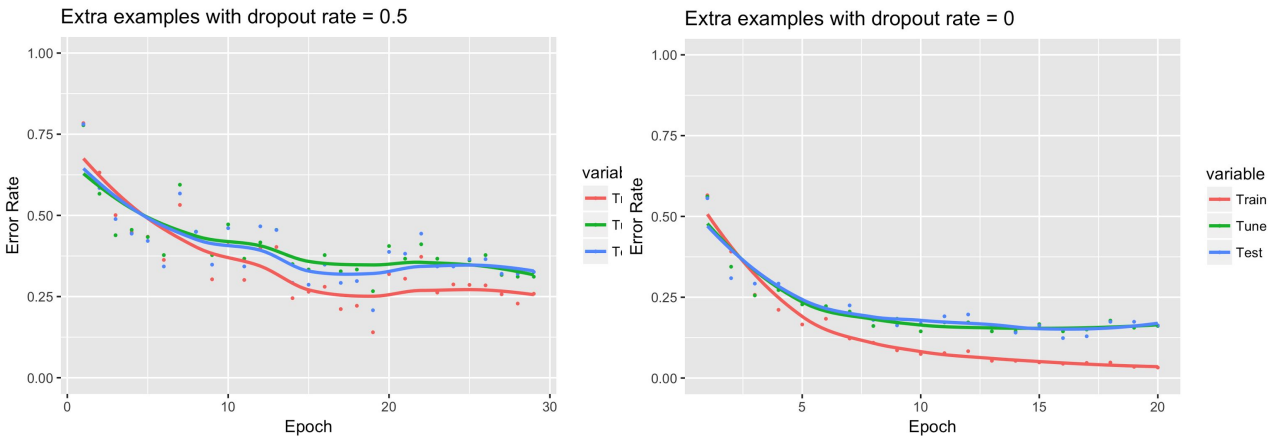
3. Learning curve with extra examples



Since we knew CNN required a large number of training examples, so we also used shifting and rotations to generate more training pictures. The effect of extra example was very significant. We could see the error rates converged very quickly (only need about 10 epoch) and hits a very small error rate of 16.85% at epoch 10.

Dropout

We also implemented the dropout technique in our CNN. We did a control experiment on whether using dropout (dropout rate = 0.5) or not with eta = 0.1.



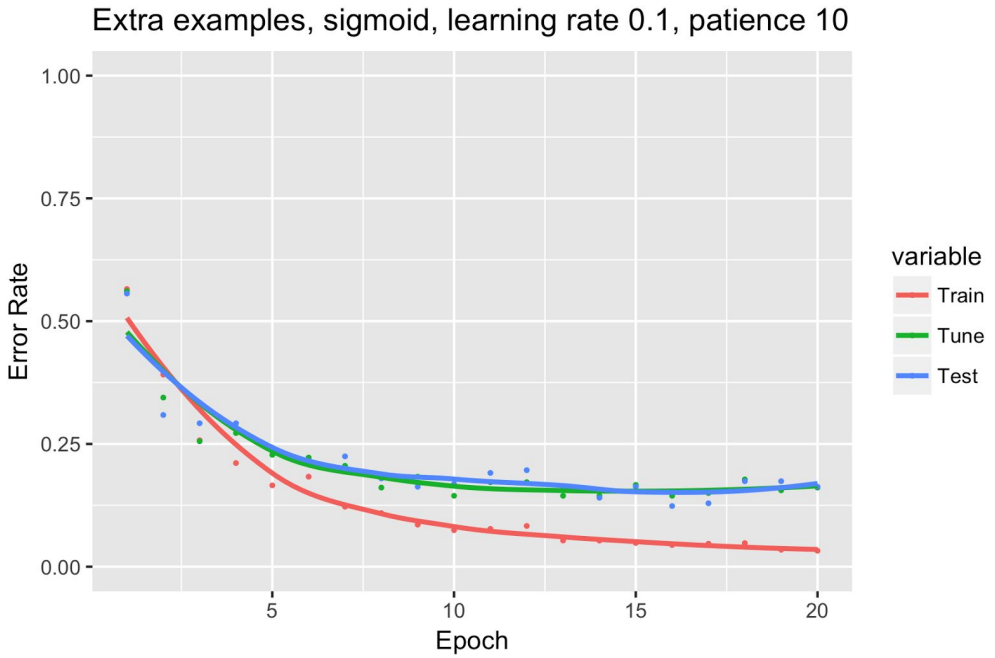
As we could see, the error rates' differences between training and tuning were larger without dropout than with dropout. Therefore the CNN had similar performance in training set and tuning set when we implemented dropout, while without dropout, the performance differed. So using dropout would effectively decrease overfitting. Since there was randomness involved in dropout algorithm, we could see the error rate fluctuated a lot for all three sets with dropout. Unfortunately dropout did not really help decrease the final test set error rate compared to the negative control group in this experiment. The best test set error rate with dropout was 20.78% at epoch 19 while the one without dropout was 16.85% at epoch 10.

Final Configure and Result

After all experiments with different techniques, we choose the following configure:

- Sigmoid in every layer
- Use extra examples (5680 in total)
- Patience = 10, learning rate = 0.1
- No dropout (dropoutRate = 0)

Here is the final learning curve, confusion matrix and error rate:



	airplanes	butterfly	flower	grand_piano	starfish	watch	predicted sum
airplanes	40	1	1	1	0	1	44
butterfly	0	14	2	1	1	2	20
flower	0	2	25	0	1	1	29
grand_piano	0	0	0	17	1	2	20
starfish	0	0	5	0	12	0	17
watch	1	1	4	0	2	40	48
actual sum	41	18	37	19	17	46	

Took 24 minutes and 42 seconds to train. Test set error count at best tune set error is 30 (error rate = 16.85%) at epoch 10.

Group Work

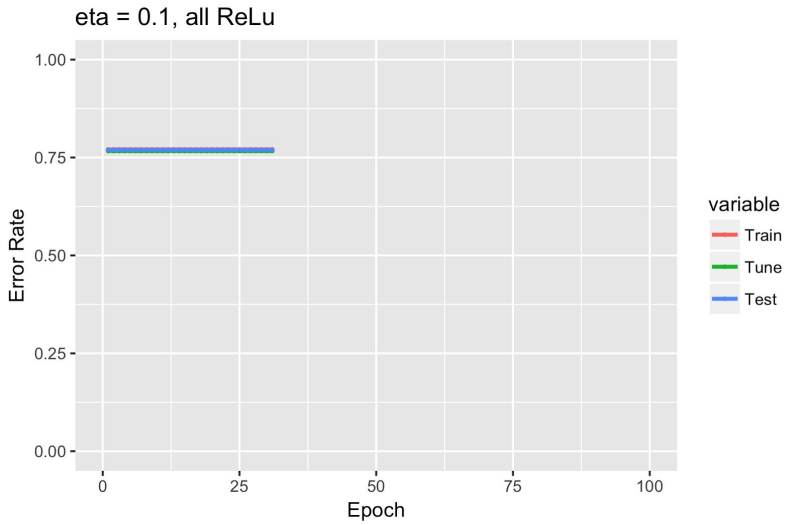
We first wrote two implementations of CNN individually. Feng's code has an unstable learning performance, while Jay's code makes some weights diverge to infinity. Then we together (literally sit together and worked on one machine) wrote and debugged the third

version. Finally in the experiment stage, Feng was focusing on dropout and Jay was focusing on extra examples and other parameter settings. Both of us have a solid understanding of the materials.

Appendix

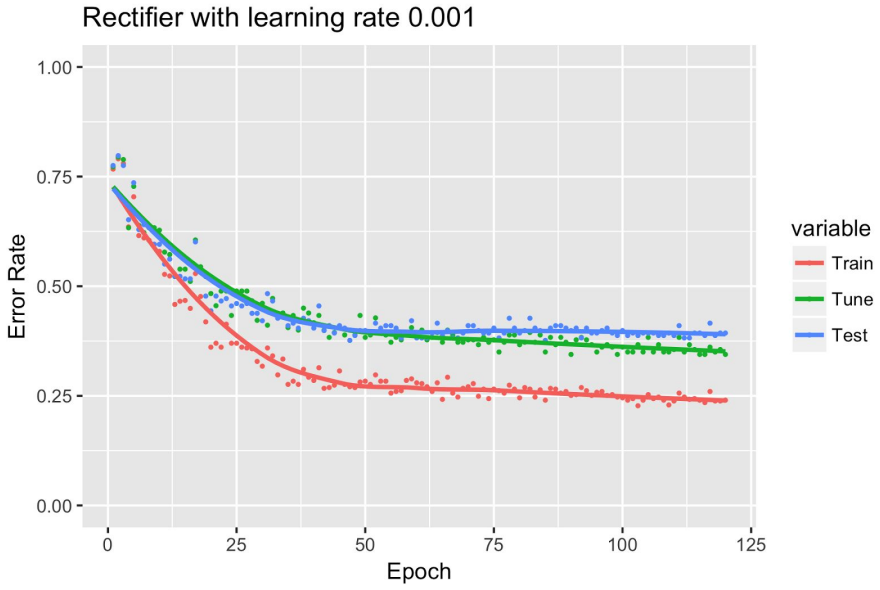
After debugging, we did more than 20 experiments to find the best configure to minimize the error rate. Among all experiments we will discussion some interesting cases In this section.

1. **Use Rectifier activation function with 0.1 learning rate**



Probably due to the overshooting problem of the Rectifier with high learning rate of 0.1, it stucked on the 0.75 error rate, and cannot learning from the training examples.

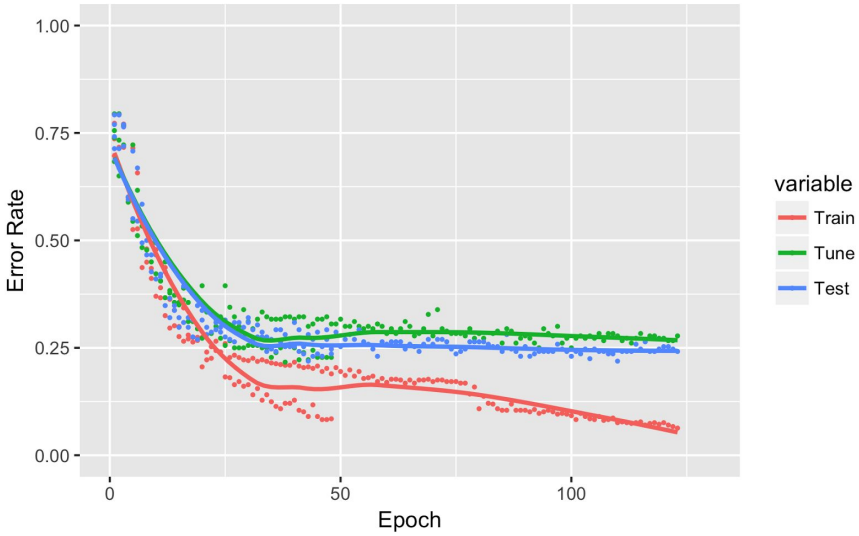
2. **Use Rectifier activation function**



After changing learning rate to 0.001, it started to learn and didn't have the overshooting problem.

3. Use Leaky Rectifier activation function

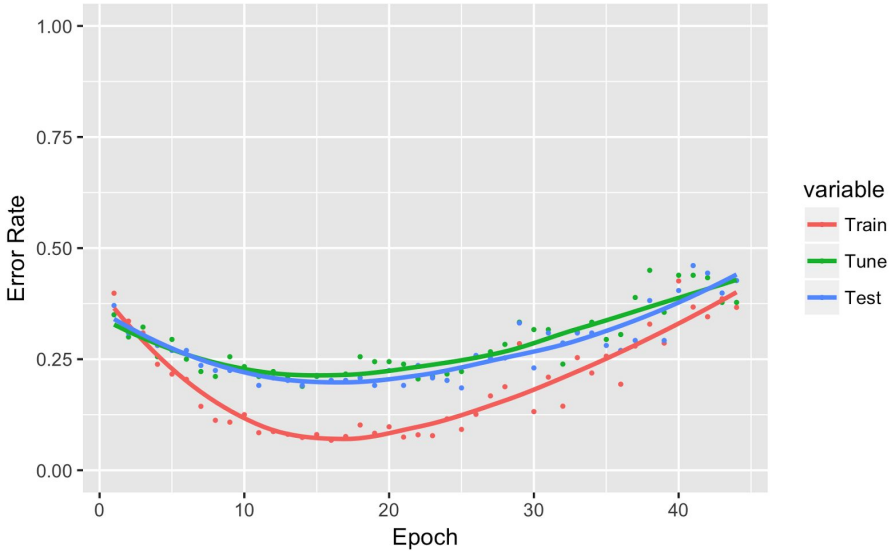
eta = 0.001, leaky rectifier activation



We could see that the overfitting problem occured at about 50 epoch. Using Leaky Rectifier activation function was faster than sigmoid.

4. Extra Example with Leaky Rectifier

Using extra examples (5680 in total) with leaky



This graph also had a very interesting finding. Instead of overfitting, the error rate bumped up again after hitting the local optimal error rate. We concluded that it was due to the special feature of leaky rectifier activation function. It was easy to overshoot, while in the later learning process the changes of weights should be small since the network was converging to the local optimal. However, the leaky rectifier kept pushing the weights and escaped the local optimal.