

Introduction

The goal of this project was to attempt to fix the perceptron's problem of relying too heavily on small features on a data set. The approach to fix it was vaguely based on the idea of adversarial neural networks, which is where one network generates fake versions and another network has to learn to tell the difference between real, and fake generated examples. Though instead of using a neural network to generate the fakes, I used a canvas that would randomly have its pixels adjusted. This means that as the training went on, the perceptron would get better at telling the reals from the fakes, but also that the fakes would become more convincing.

Setup

I implemented 3 different trainers for the perceptron. The first was the normal perceptron that we learned about in class; use a training set. Test against the perceptron. If the perceptron gives a wrong answer, then adjust the weights.

Next I implemented my new version, where the trainer is given examples of the things to match and then mutators. The mutators, as mentioned earlier, are trained alongside the perceptron. After each training phase, all mutators will randomize their weights a little, and if they score higher than they would have before, they keep their new weights. Other than that it is the same as the original trainer.

Finally I made a trainer that is a mix of both. Half of the wrong examples are real, and the other half are fakes generated by the mutators.

For training data, I decided to use the simple and recognizable MNIST data set, downloaded from <http://yann.lecun.com/exdb/mnist/>.

Methodology

To test the trainers against each other, I initially wanted to train up the perceptrons over a decent period of time, but I quickly found that a single perceptron trains very fast, and has inconsistent swings in performance if it has a bad training period. So instead of training under each trainer for a while, I decided to use the iteration of the perceptron that scored the highest on the test set over the entire training period for each trainer. Because of this, my results don't mean much in terms of how a real perceptron would be trained, but since I am just looking at the trainers relative to each other, this is fine.

Next, the test I decided on was taking the best perceptron from each trainer and then putting that against the remaining unused 9,000 image MNIST testing set. From this, I measured the percentage of correct positives and negatives each perceptron identified out of all positives and all negatives.

In order to normalize the results a little, I ran the above process 5 times and then averaged the results.

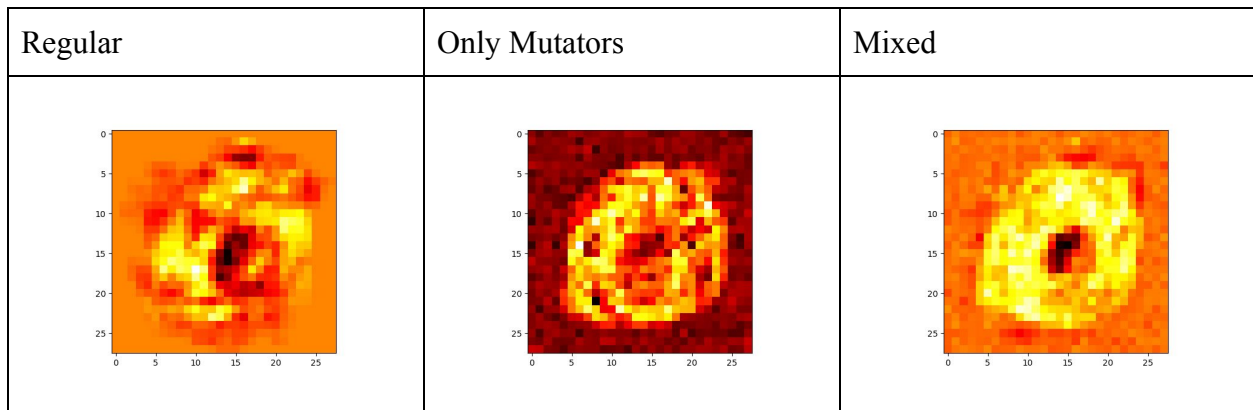
I also took out the pattern on the perceptron for each trainer after a smaller training period, along with some of the patterns that were generated by the mutators.

Results

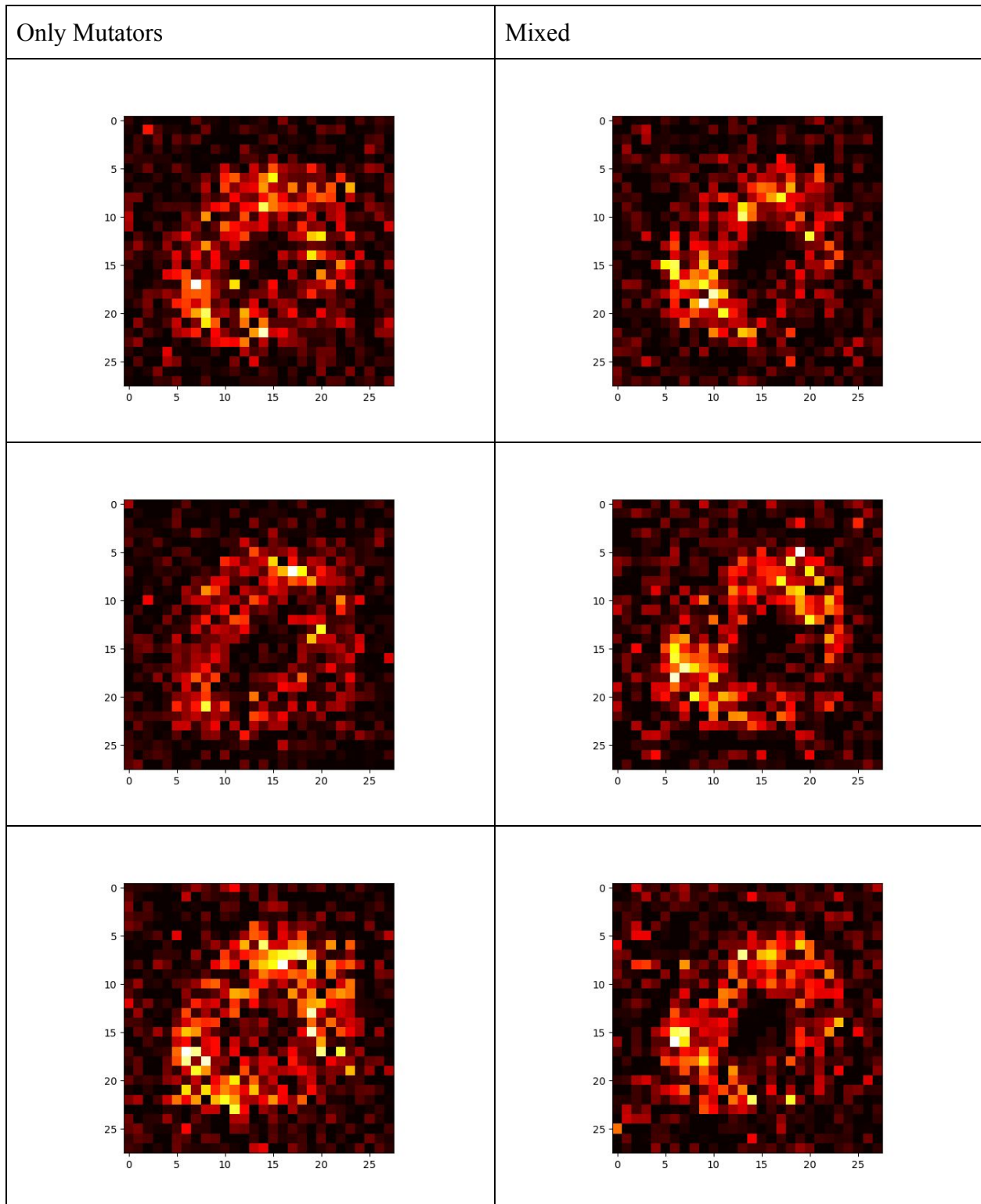
Table of Average Success Rates for the Different Trainers

	Regular	Only Mutators	Mixed
% positives identified	98.9	97.8	97.1
% negatives identified	96.0	72.1	97.3

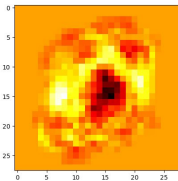
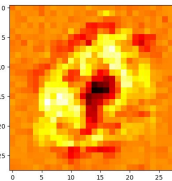
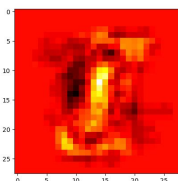
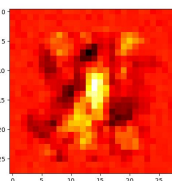
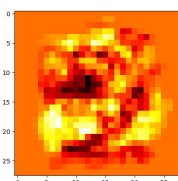
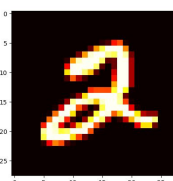
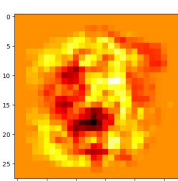
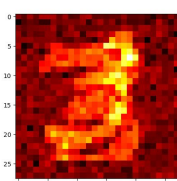
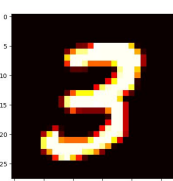
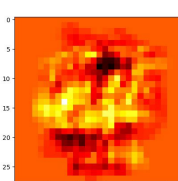
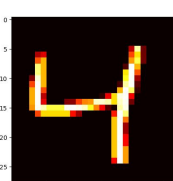
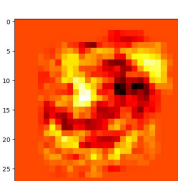
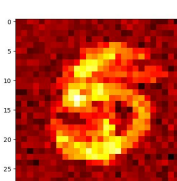
Perceptron Patterns with a Long Train Time for 0

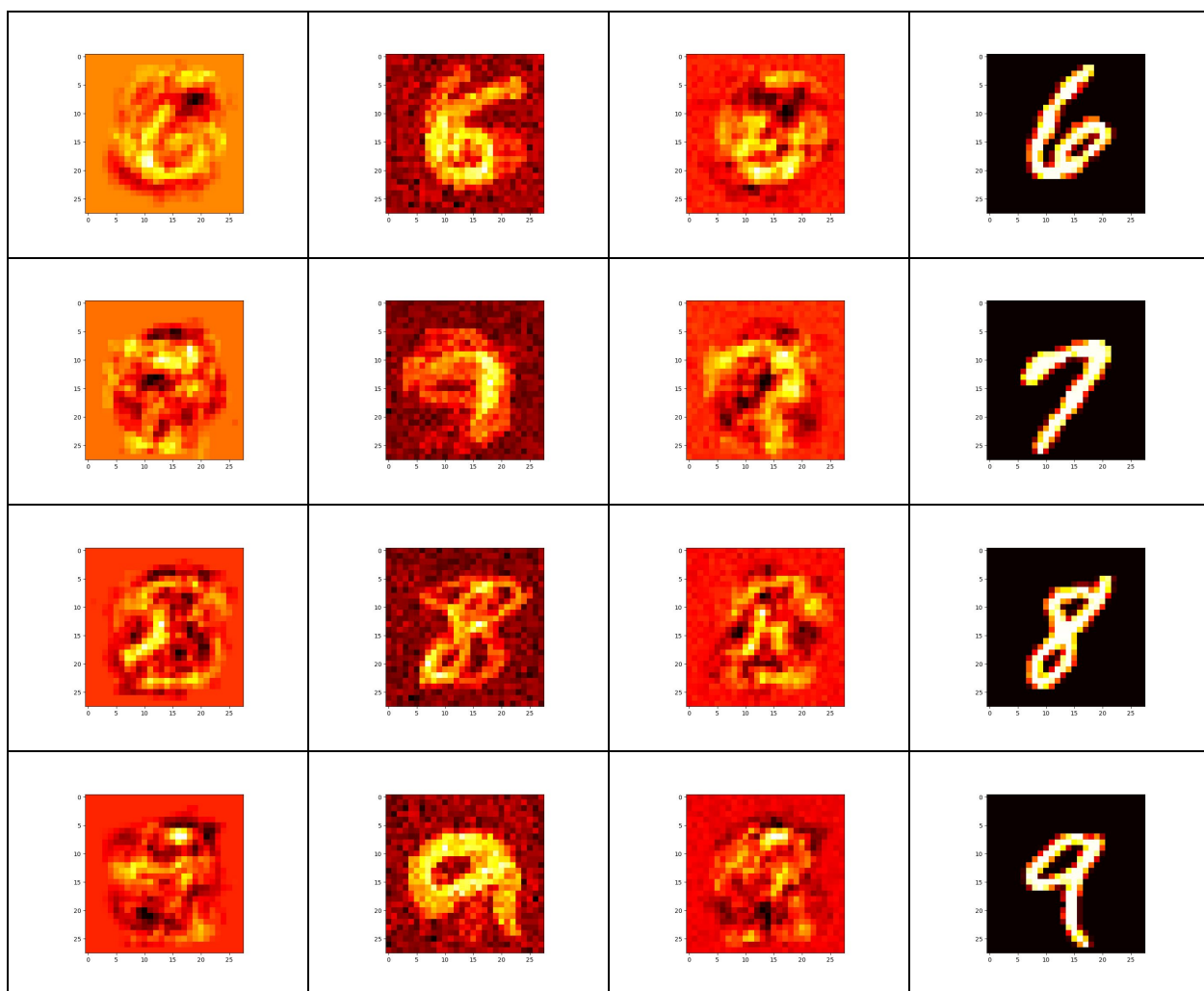


Sample of Mutator Patterns



Perceptron Patterns for all Numbers at Reduced Training Time

Regular	Only Mutators	Mixed	MNIST Example
			
			
			
			
			
			



Discussion of results

Unfortunately there doesn't seem to be any advantage to using the mixed approach over the normal approach, and the only mutators approach is noticeably worse. At the very least I can say that I aesthetically like the look of Mixed trainer's perceptron more than the Regular one, as I think it looks more like a 0.

I don't have much to say about the mutators except that they look very similar across both trainers. I looked at more of them outside the 6 examples I gave, and I still couldn't see much of a pattern.

It seems that for shorter training times, the "Only Mutators" trainer can afford to over fit the data, as its mutators won't be accurate enough yet to contest with the real examples. Also the mixed seems to lose a lot of definition.

Something that I didn't find interesting enough to make a graph about but I think is important is that all but one of the success rates for both categories and all trainers rose very fast and stayed up for the most of the time. The exception was the % negatives identified by the "Only Mutators" trainer. Instead of rising quickly, it would very slowly climb over a much longer period than everything else. This is explainable, as that is the only category that doesn't have direct access to a data set similar to the one that it will be tested against, unlike every other category, because the "Only Mutators" trainer isn't given any non-generated negatives.

Conclusion

Overall I'd say it was a mild success. I was able to implement perceptrons and make them a little more recognizable. My main problem was that the mutators took too long to catch up with the perceptrons, but I was tentative to speed up the mutation too much because then the mutator might start looking too close to the number it is trying to copy, to the point that it wouldn't be useful to train against. I don't think that I've refined perceptrons though, the regular implementation seems to work just as well, and also runs significantly faster.