Project 1

Parsa Merat (217554197)
merat@my.yorku.ca

November 27, 2024

Something to note, in my notebook, it automatically limited the output that was shown on screen. If that doesn't happen on your screen, please just fold the outputs....

1 Logistic Regression

I did not do anything fancy here. The algorithm is pretty straight forward. Used full data gradient descent. I needed to play around with the learning rate a bit so it locates the local maxim better. Used a simple method of reducing the learning rate every epoch (multiplying by 0.99) which yielded a 0.924 training and testing accuracy. The algorithm ran for 200 epochs (but stablised around 50). Finished in less than 10 seconds. Some cute plots can be found in the jupyter file.

2 Ensemble SVM with RBF kernel

best configurations for the ensemble, this configurations was fixed for all SVMs in ensemble:

kernel	C	gamma	train accuracy	test accuracy	average number of SVs
linear	100		96.096	94.740	531
rbf	10	9	99.163	96.920	791

SMO vs PGD

For SVM rbf (7 vs 9) for gamma=9, C=10:

10 epochs of PGD (lr=3 showed fastest convergence) resluts in a loss of around 1820 in 33 seconds. it takes SMO (counter=700) 18 seconds to reach the same loss (with very simmilar train and test accuracy of 98 and 97, since they have the same loss). SMO and PGD both have around 2500 SVs

So to reach the same loss (accuracy) without fully converging, SMO is twice as fast. They end up with the same number of SVs.

However to fully converge in both cases:

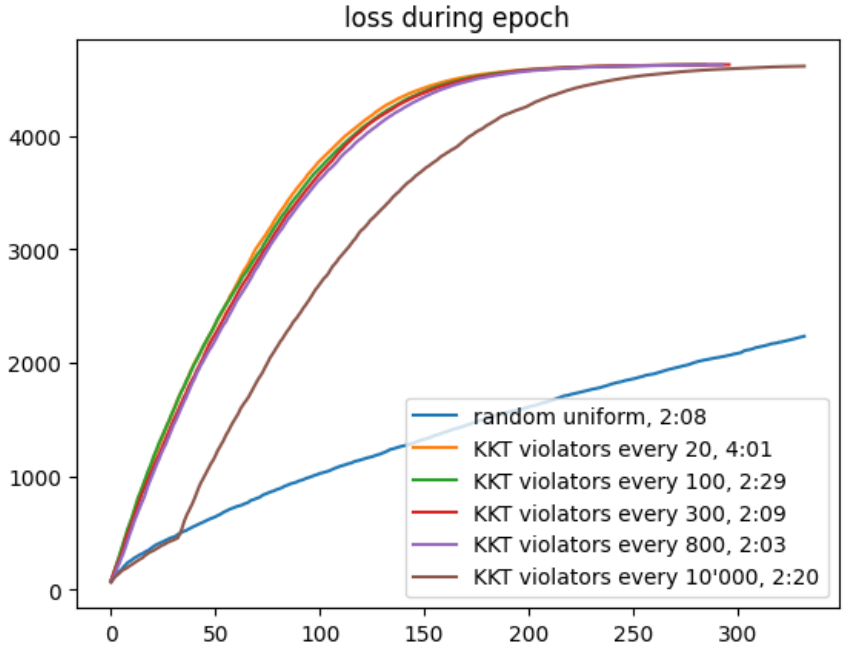
optimiser	loss	train accuracy	test accuracy	number of SVs	time
PGD	4312	99.33	97.89	2388	4:50
SMO	4628	99.43	97.99	1444	2:07

In terms of speed, SMO seems to be at least twice as fast as PGD. At full convergence, with better convergence (less loss) and better testing accuracy and much less number of SVs Also no need to find hyperparameters like learning rate, annealing etc.

I tried my best to vectorise everything so that the calculation of solving the quadratic and clipping α_i and α_j would be as fast as possible. I countered many problems that Im going to list and explain what I did to manage them:

Choosing i and j

At first i decided to randomly choose i and j. This method was easy and fast to give out answers. But the training per iteration was very small. The next method who showed more success wad to chose i from αs that violated the KKT condition. When choosing i, I gave more priority (by increasing its probability) to violators that where not 0 or C, since they are the "more important" support vectors. Next step is to choose j. I noticed if I also choose j from this set of violators, it converges faster (rather than choosing j uniformly from all αs). But to find these violaters accuratly, I need to calculate the prediction of my current model on the whole data set every itteration (since changing any alpha might affect this), which would be very costly (or maybe there is a way to update the prediction at every itteration based on changes to α_i and α_j? I think I jumped to coding too soon, I should've spent more time on paper). So it might be a better idea to calculate these violators every m epoch. but how to choose this counter for a good time/performance trade-off? Test and look for the best one



It seems that at the very start, violators change more dramatically. So maybe its a good idea to check them more frequently at first and slow down later. But for now, 800 seems to be fast enough at converging.

Another way of choosing j (instead of randomly from a set) would be to find a j that maximizes the derivetive of our objective with respect to α_i, so that when α_i is updated using:

$$\alpha_{extreme} = \alpha_i - \frac{(-Q_i + y_i y_j Q_j)\alpha + 1 - y_i y_j}{-Q_{i,i} - Q_{j,j} + 2Q_{i,j}}$$

it would result in the biggest change of alphas (however clipping is not takan in to consideration). Although promessing, I did not have much success with this approach. Since a small number of predictions would grow in value and kept being picked most of the time. And although they would technically give the biggest step, since α_i needed to be clipped, these steps would not actually be made. The next step would've been to not select the previously chosen i and js, but I did not furthur pursue it (as the previous method was satisfactory).

Choosing hyper parammeters

My algorithm would keep running until the objective converges, so number of epochs is irrelevant. For gamma and C (in rbf kernel), I found the pair with the highest test accuracy using simple grid search and used that to train all the other SVMs. For C in linear kernel, similar search approach.

Voting

I trained the ensemble SVM model on the data and used a simple voting to determain the final class. I noticed that on images that it missclassified, the difference between the votes of the right class and the wrong class where quite close. My thought process was to get rid of the votes of unrealated SVMs that might be sabotaging the true class. I chose the classes with maximum votes, and maximum-1 votes and applied the voting again just on those. If there was a unique maximum, it was returned. However if the maximum spot was tied, I would reapply the algorithm to the maximums (throwing away other voters). I would do this recursivly until either a max is found, or all the votes are tied. In the latter case, return randomly.

This algorithm did not help much, so I returned to the simpler original :) My other idea to solve ties was to give weight to the vote of every SVM, based on the distance of a point and the SVM boundry (and soft boundry). Which I did not pursue.

3 Non linear feature extraction

I first find the best network to extract a certain size of features. For auto encoder, this ”best” means an encoder that minimizes the test MSE. For bottle neck, it should maximise test accuracy (minimse CE). Then I compare the performance by finding the best SVM for each feature extractor (the result can be seen later).

Autoencoder

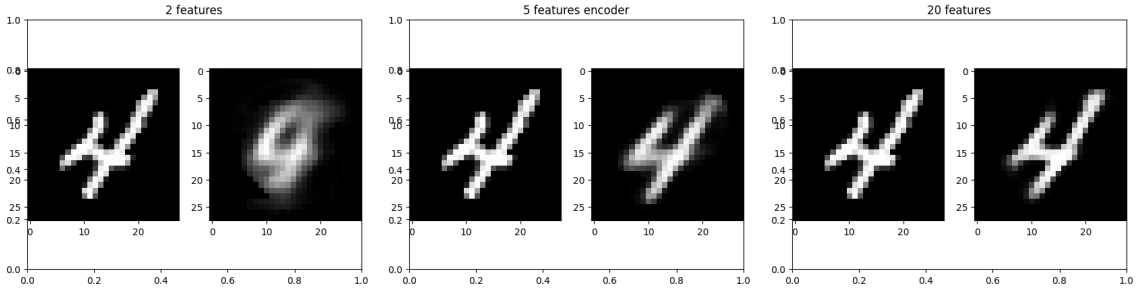
For the auto encoder, I changed the final activation function to either sigmoid or RelU (since softmax doesnt really make sense) (also changed the backpropogation accordingly).

I noticed I get the best result (least test MSE) using networks of structure [1024,x,1024] with a sigmoid output function. If my network got deeper, it would not train very well which I believe is because my optimizer is too simple.

I also noticed that if I normalized every pixel to have L2 norm of 1 (like it was done for SVM) my auto encoder would not achieve lower MSE rather than if i didnt. So i did not normalise the pictures before feeding them to auto encoder.

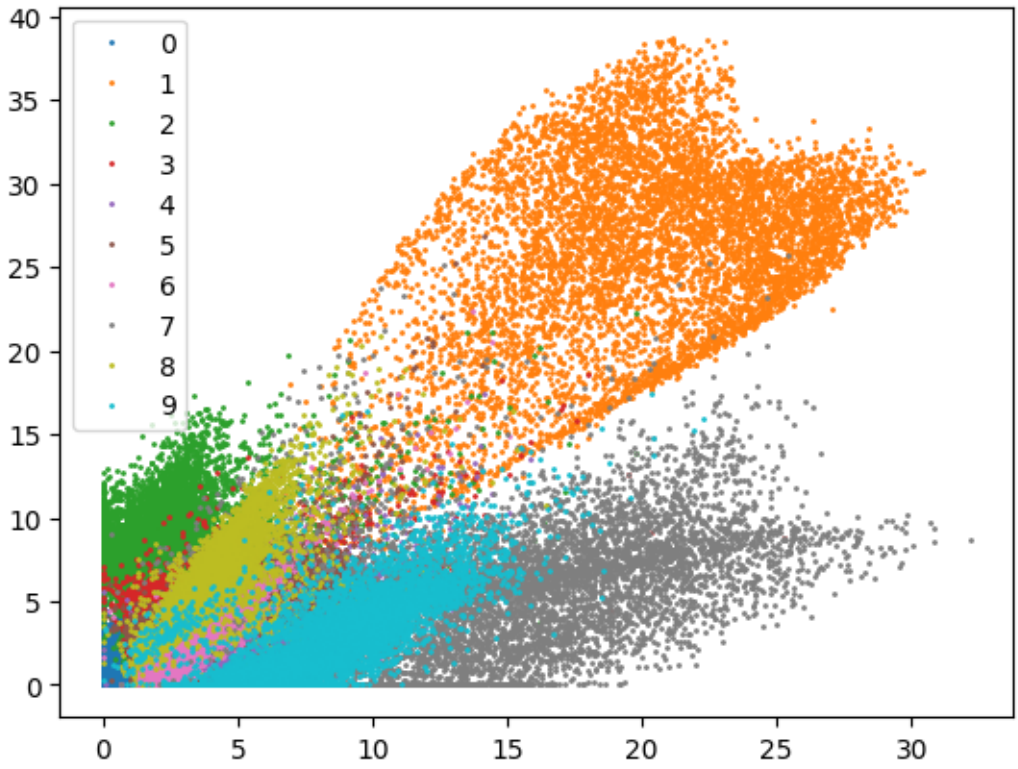
feature dimmensions	train MSE	test MSE
2	33.1191	33.4957
5	19.5659	19.9052
10	10.5064	10.7964
20	5.5982	5.8835
40	3.1757	3.3851
80	2.1254	2.2436

Heres some cute plots of Autoencoder encoding and decoding pictures that shows the loss of data.



I then found out that SVMs with gamma=0.01 and C=10 gave me the highest test accuracy (by grid searching with svms on features of size 20). keep in mind that maybe for different number of features, this might not be the ”best” svm. But its somehow fair if we use this accros all SVMs for autoencoders.

Heres also a cute plot of what happens when we enocde to a 2D plane:



One thing that might have been good to try was to use different activation functions for the encoded layor. Maybe then it would've been less dense at 0,0.

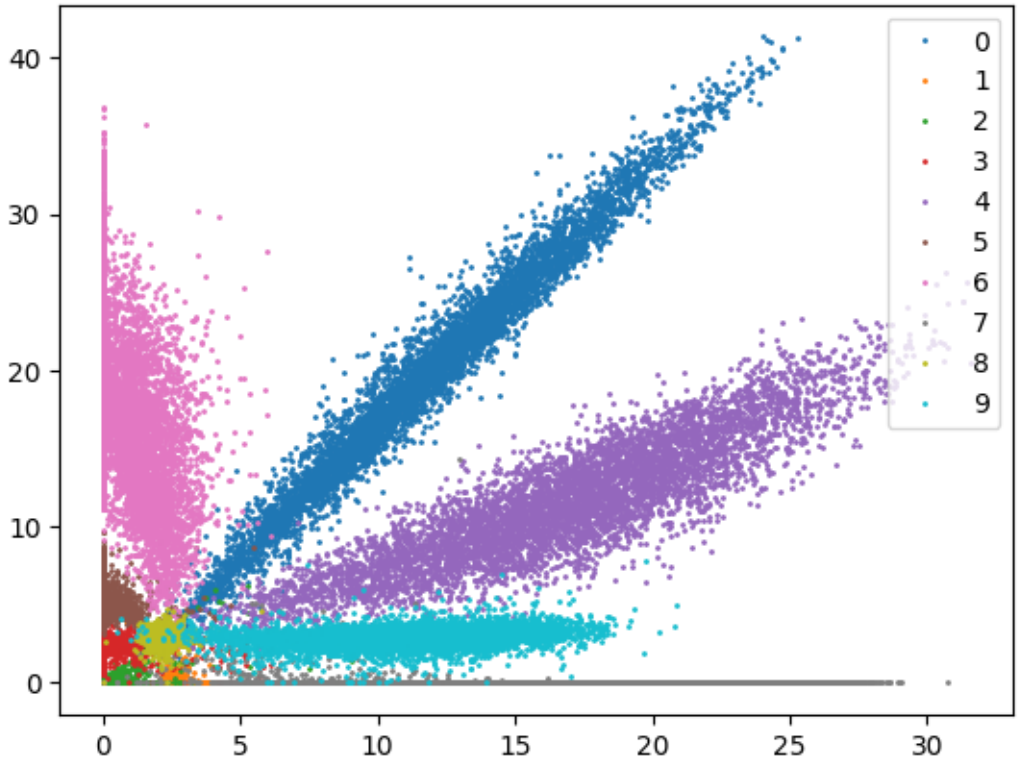
Bottleneck

For feature size of 20, struct of [500,20,50] gave me the highest test accuracy. I will be using [500,x,50] to extract the features of size x.

feature dimmensions	bn train acc	bn test acc
2	98.1	95.51
5	99.87	98.01
10	99.96	98.33
20	99.97	98.47
40	99.98	98.48
80	99.99	98.50

For feature size of 20, SVM with gamma=0.01 and C=10 gave me best result. So Ill use these hyper parameters to test every encoder.

And heres what happens if we extract 2 features:



Bottleneck vs Autoencoder

7 vs 9

	bottle neck		auto encoder	
feature dimmensions	test accuracy	#SVs	test accuracy	#SVs
2	98.43	641	85.22	4051
5	99.26	43	93.62	3735
10	99.02	39	98.63	771
20	99.41	46	99.21	1127
40	99.66	53	99.41	1760
80	99.41	101	99.31	2027

0 vs 8

	bottle neck		auto encoder	
feature dimmensions	test accuracy	#SVs	test accuracy	#SVs
2	99.03	93	96.01	1994
5	99.54	25	98.98	421
10	99.85	23	99.64	282
20	99.90	21	99.59	825
40	99.74	37	99.74	1525
80	99.74	50	99.64	1648

One thing to note is the very small number of SVs necessery for bottle neck. which makes sense since the whole point of bottle neck (a non linear continuation of LDA) is to place same classes closer to eachother. So its easier to find a hyperplane. keep in mind that for regular rbf svm, the best that we managed to achieve without feature extraction was testing result of 97.938 for 7 vs 9 and 98.976 for 0 vs 8.

In general it looks like bottleneck performs better. However, I feel the neural network in bottleneck is partially solving the problem without SVM which....feels like cheating.