Machine Learning Handin 2

Lasse Melbye & Søren Elmely Pettersson

October 2014

1 SVM

1.1 Data Generation

The `AuTrain.mat` data set is quite small, compared to the dimensionality of the images. To combat this we tried generating more data by adding various forms of noise to the data we already have. This is handled by the matlab function `preprocessSvm.m`. One approach we did was to take the entire dataset, and add gaussian noise with *imnoise* to each picture, and afterwards concatenate these "new" images to the existing dataset, doubling the data size. A similar thing was done by adding rotation to the entire dataset, and afterwards concatenating these "new points", also doubling the data size with these new points. The rotation was taken randomly for each picture in the interval of $[-40, 40]$ degrees, which was the same range as we found Le-Net to be using - higher values seem "unrealistic" for real handwritten numbers.

To see if this preprossing had any effect, we choose some arbitrary fixed values for the hyper parameters, and trained an SVM for each of the kernels, for each of the possibilities of our data. We set aside 1000 images of the data as test data for the remaining part, which we trained our SVMs on. This was for the AuDigits only. We made sure that the test data was not a part of the data we made processing on. Below are some of the results summarized:

Linear Kernel:

Data	Datasize	cost	acc AuTest
original	4982	0.01	87.8%
gaus noise	9964	0.01	88.2%
rotation	9964	0.01	85.9%
rotation + gaus	14946	0.01	87.5%

Poly Kernel:

Data	Datasize	cost	dimension	acc AuTest
original	4982	200	2	91.2%
gaus noise	9964	200	2	91.9%
rotation	9964	200	2	91.0%
rotation + gaus	14946	200	2	91.9%

RFB:

Data	Datasize	cost	gamma	acc AuTest
original	4982	200	$\frac{1}{feat}$	90.7%
gaus noise	9964	200	$\frac{1}{feat}$	91.3%
rotation	9964	200	$\frac{1}{feat}$	89.1%
rotation + gaus	14946	200	$\frac{1}{feat}$	90.4%

It would appear as though the classification, and error estimate, actually drops slightly. If we generated even more data, it would probably fall even further. We still decided to use this generated data as train data, to prevent the fitting of noise and the degrees of freedom, when trying to optimize further, in the next sections, just from the fact that there are more data points.

1.2 Cross validation and selection of hyper parameters

To find the best kernel parameters we used grid search over combinations of the parameters. For each combination, we used 5-fold cross validation, on our data we generated for AuDigits as explained above, and searched for the best parameters for each kernel. We incremented the different parameters by an exponential factor for each iteration of the grid search, as recommended in the `libsvm` guide. After the cross validation, a model was trained on the entire training set with these found parameters. Afterwards, this model was tested on the set-aside test part of our data for AuDigits.

To summarize, (i) find optimal parameters using cross validation, (ii) train model using these parameters on full training set, (iii) test produced model on test set. Results are summarized below.

1.2.1 AuDigits

The grid search for the different kernels are all on the generated data we have for the Au digits set. For each of the following, we first set aside 1000 images for testing after the grid search - this data is not used in the cross validation, or anywhere else, only for the final testing after the grid search for the best parameters have been done. More data could have been set aside for testing, but since we know Allan is hiding only 1000 images, we wanted to use more for training, at the cost of a worse out of sample estimate.

For the *linear kernel*, there is only one hyper parameter, namely the cost C, so this was more of a linear search, as opposed to the two other kernels, which used grid search. The interval for the cost was set to $[2^{-3}, 2^{11}]$. Results:

Data	Best Cost	cross val	autest	mnist
Au+rot+gaus	32	85.82 %	81.2 %	24.15%

For the *Poly Kernel*, the initial grid search was in the intervals of $C \in [2^5, 2^{11}]$ and $d \in [2, 8]$ which produces the following results:

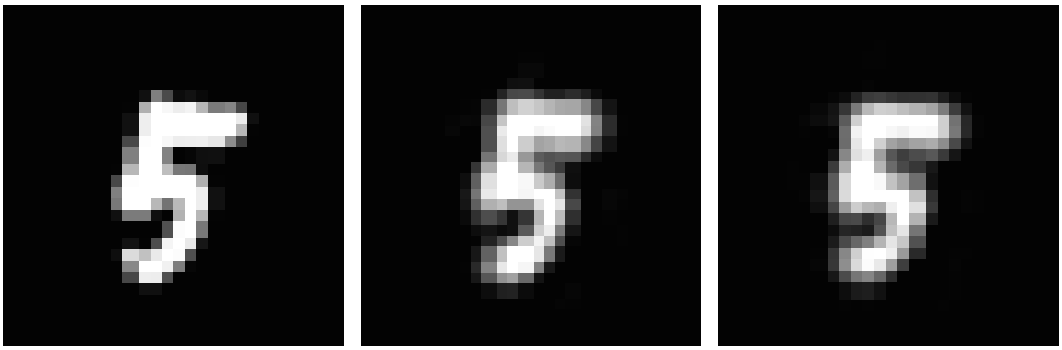
Data	Best Cost	Best d	cross val	autest	mnist
Au+rot+gaus	2^{11}	2	95.05 %	93.2 %	N/A%

It appears that the area we are searching in is too narrow, the dimension appears to be locked at the initial 2, but the cost is maximized. We thus expanded our search for the cost to the interval of $[2^{11}, 2^{19}]$ and obtained the following:

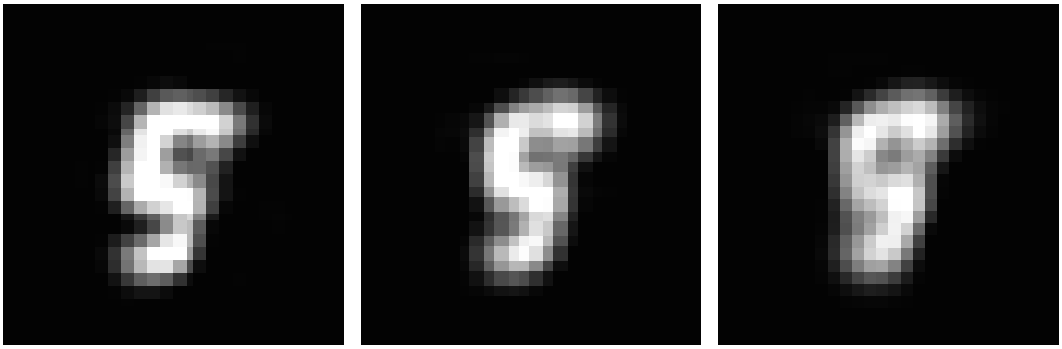
Data	Best Cost	Best d	cross val	autest	mnist
Au+rot+gaus	2^{15}	2	95.57 %	93.8 %	35.38%

For the `RFB kernel`, the values for cost where sought for in the interval $[2^{-3}, 2^5]$, and the values for $\gamma \in [2^{-10}, 2^{-2}]$. Results:

Data	Best Cost	Best γ	cross val	autest	mnist
Au+rot+gaus	2^5	2^{-4}	97.39 %	95.3 %	41.46 %



(a) Original (b) Dimension 100 (c) Dimension 75



(d) Dimension 50 (e) Dimension 25 (f) Dimension 10

Figure 1: A 5 after dimensionallity reduction

Again, we keep searching for a better cost, since it appears to be maxed. We thus lock the γ and expand C to $[2^5, 2^9]$. $c = 2^5$ still came out on top.

It would seem that we are prone to overfitting in the above - our estimate found using cross-validation on a set of hyperparameters, does not track the out of sample estimate, of the set aside test set, very well. Note that we could have searched even finer in the regions where the best cross validation appeared, instead of taking steps of the second exponents, we could have searched finer, but due to time, we did not get around to it.

1.3 Using PCA on AuDigits

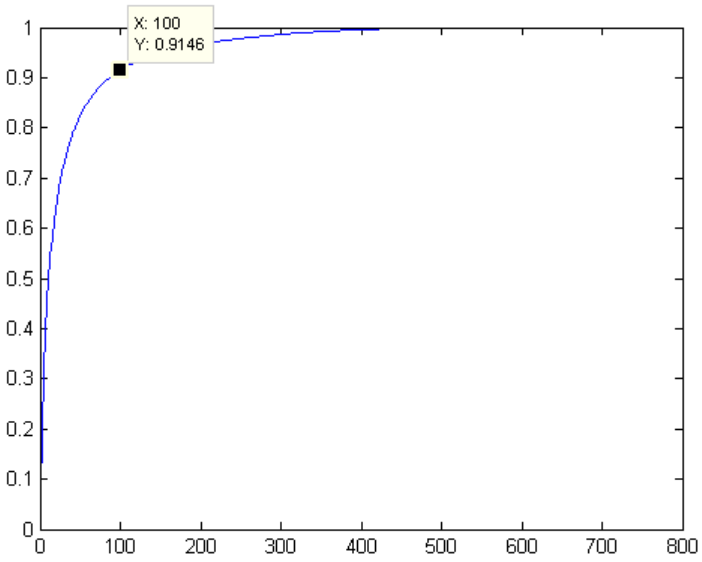
We tried reducing the dimension of our in data. The dimensionality reduction was done on preprocessed data in which points which were rotated randomly between $[-40, 40]$ degress were added. Afterwards the dimension was reduced and training was done on the the new data. Prediction was done on test data dimensionality reduced to the same basis as the training data.

Data	Dimension	Kernel	Accuracy
Au+rot	100	RFB	79.4 %
Au+rot	300	RFB	91.1 %
Au+rot	300	Poly	92.3 %

We tried constructing a test image after its dimension had been reduced. This can be seen at figure 1

1.3.1 Mnist

For the mnist digits, we once again wish to utilize the grid search to find the best hyperparameters. But, this "exhaustive" search (i.e non-clever search) takes quite a while, and now we have a wooping 60.000 images. We thus decided to do PCA on the mnist digits, as to lessen the dimensionality of the images. By utilizing matlabs built-in `princomp()` and plotting the cummulative sum of the eigenvectors `cumsum(latent)./sum(latent)` we obtain the following plot of the variance of the different principal components:



The first 100 principal components out of the 784 account for 91.46% of the overall variance. We thus decided to use the 100 first principal components of the training data, in the grid search. The test data went through the same PCA analysis, i.e we took the 100 first principal components, and took the coefficients from the `princomp` of the train data, and multiplied onto the testdata: zscore(Test)$\cdot$(coeff$_{PCATrain}$), to express the testdata via the same basis. The results are summarized below.

`Linear Kernel`

Data	Cost	cross val	autest	mnist
PCA 100, 20k images	2^{-3}	93.57%	N/A	89.15 %

For the `Poly Kernel`, the grid search was in the range of $C \in [2^5, 2^{11}]$ and $d \in [2, 6]$:

Data	Best Cost	Best d	cross val	autest	mnist
PCA 100, 20k images	128	3	97.79 %	N/A	96.18%

`RFB Kernel`: Here the cost C was chosen over the interval of $[2^{-3}, 2^5]$, and $\gamma \in [2^{-10}, 2^{-2}]$:

Data	Best Cost	Best γ	cross val	autest	mnist
PCA 100, 20k images	2^3	2^{-6}	97.81 %	N/A	33.22 %

The last result for the RFB kernel of 33.22% for the out of sample is really weird, considering the cross validation error. We did not discover the reason for this.

We tried retraining the original data, without PCA reduction, with the found hyper parameters and achieved the following results (this shouldn't necessarily give good results):

`Linear Kernel`

Data	Cost	cross val	autest	mnist
mnistTrain	2^{-3}	N/A	N/A	94.59 %

`Poly Kernel`

Data	Cost	d	cross val	autest	mnist
mnistTrain	128	3	N/A	N/A	95.85%

`RFB Kernel`

Data	Best Cost	Best γ	cross val	autest	mnist
mnistTrain	2^3	2^{-6}	N/A	N/A	98.44%

It seemed to perform rather well, even though the hyper parameters where found for the images, expressed in another basis, and not all of the components where even considered. Point being, these hyperparameters should not be optimal for this case, but appear to give good results none the less.

1.4 SVM Best Classifiers

Since the overall goal was to do the best possible on the hidden 1000 images from AuDigits, we seem to have a clear winner with regards to the different produced SVM classifiers from the above sections, namely the RFB kernel trained on the generated extra points from AuTrain with added rotation and gaussian noise, trained with the parameters $C = 2^5, \gamma = 2^{-4}$. The estimate for the out of sample accuracy is $95,3\%$ which is what the trained classifier achieved on our set aside test set of 1000 images (this size was chosen due to the hidden images having similar size).

This classifier is saved in the requested `bestSVM.mat`, and this will be loaded and predict on n images by `runBestSVM.m`, which returns a vector of predictions.

2 Sparse AutoEncoder

The first thing we tried to distinguish was whether to feed the features into an SVM or softMax. We grabbed the AuTrain as well as our own generated extra points with gaus and rotation:

model	Data	hiddenLayer	sparsity cost	test
softMax	auTrain	200	0.1	92.60 %
SVM RFB	auTrain	200	0.1	96.16 %
softMax	gaus+rot	200	0.1	77.65 %
SVM RFB	gaus+rot	200	0.1	85.95 %

We see a significant difference between using the original AuTrain.mat and our own generated datapoints for AuTrain with added noise and rotation - the latter performed significantly worse. The SVM RFB kernel was taken with the optimized hyperparameters from the previous section (they where found for other data, and not these features, but we thought there might be a relation hidden in there somewhere to make the parameters good here as well). It seemed to outperform the

softMax by a large margin. We thus continue with the SVM, and on just the original AuTrain for this part, since the difference appears to be so large. One should think that the added datapoints would help, but apparently not.

The next thing we tried was manipulating the size of the hidden layer. We wanted to see the effect of enforcing actual sparcity, by making the hidden layer larger than the input size. The results for different sizes of the hidden layer are summarized below:

Data	hiddenLayer	sparsity cost	test
auTrain	50	0.1	88.24 %
auTrain	200	0.1	92.60 %
auTrain	500	0.1	94.05 %
auTrain	28·28	0.1	94.58 %
auTrain	28·28·2	0.1	95.11 %
auTrain	28·28·4	0.1	94.78 %

The results seem to generally perform better the larger the hidden layer is, with some irregularities. A hidden layer above the input size actually makes it a sparse autoencoder in the sense that the sparsity constraint enforces the hidden layers to activate less, and thus still learn the "identity" if you will. We tried continuing with the hidden layer of double the input size (28*28*2) and change the sparsity parameter:

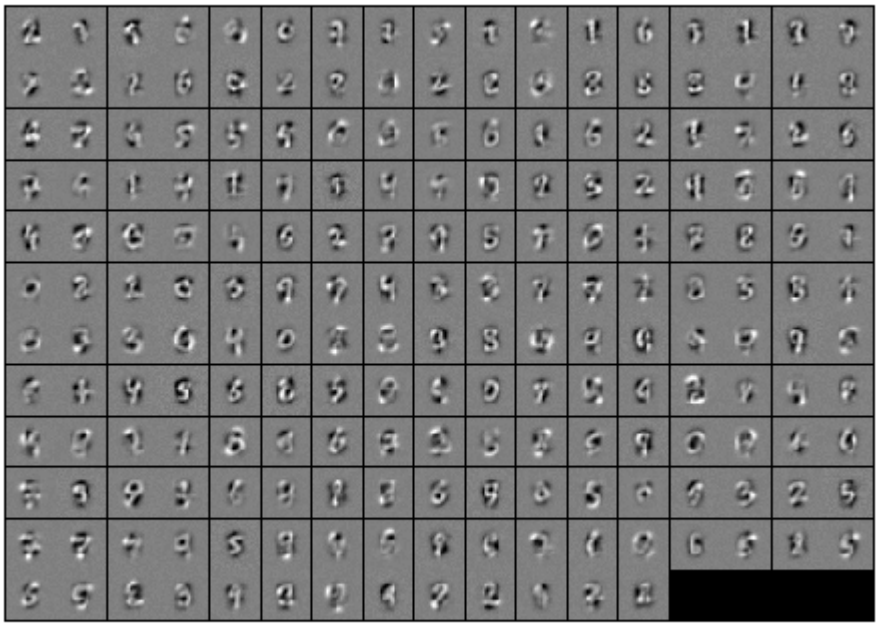
Data	hiddenLayer	sparsity cost	test
auTrain	28·28·2	0.01	89.29 %
auTrain	28·28·2	0.05	93.39 %
auTrain	28·28·2	0.1	95.11 %
auTrain	28·28·2	0.2	95.44 %
auTrain	28·28·2	0.3	95.31 %
auTrain	28·28·2	0.4	95.70 %
auTrain	28·28·2	0.5	95.31 %
auTrain	28·28·2	0.6	95.70 %
auTrain	28·28·2	0.7	95.57 %
auTrain	28·28·2	0.8	95.57 %

As soon as the sparsity surpasses a certain threshold, the effect seems to be almost insignificant. Also, none of the above values beats the initial values of the parameters in the given `testing.m` file, as can be seen (the RFB kernel with 96.16%).

Instead of training the sparse autoencoder on 5-9, and training softmax/SVM to predict 0-4, we tried splitting the entire dataset in half, and use the first half as unlabeled data. The remaining half was split evenly into respectively training and test data. We tried this for both our own generated data, as well as the original AuTrain data. We used the original parameters of hidden layer 200 and sparsity constraint 0.1. The minFunc iterations where set to a 100 as well. The results where as follows:

model	Data	hiddenLayer	sparsity cost	test
softMax	auTrain	200	0.1	88.0 %
SVM RFB	auTrain	200	0.1	90.7 %
softMax	gaus+rot	200	0.1	80.87 %
SVM RFB	gaus+rot	200	0.1	86.04 %

It seems to be "harder" to predict all the images, and not "just" 0-4, which intuitively makes sense. The visualization of the weights in the hidden layer, maximized, looks as follows:



It seems to actually resemble parts of numbers!

We tried the same on the mnistTrain set and obtained the following results:

model	Data	hiddenLayer	sparsity cost	test
softMax	mnistTrain	200	0.1	90.59 %
SVM RFB	mnistTrain	200	0.1	93.73 %

Again the support vector machine seems to outperform the softMax, and in general classifiers trained on mnist perform better (on the corresponding test set). Clearly, there are more datapoints, which helps.

One last thing we tried with the sparseAutoEncoder was mapping the features from mnist to auTrain. We ran the sparse autoEncoder on the mnist data, extracted the features, trained an SVM on these features, and tried predicting all the images in AuTrain. The result was a mere 10.03%, i.e with 10 classes/different digits, it can't really get worse, as this equals just guessing randomly uniformly. We are quite skeptical that we did this correctly, as such a fail rate seems somewhat grotesque.