CPSC 340 Final Part 1: Machine Learning with MNIST Dataset

36246163 l4y0b

1 Introduction

The MNIST classification problem is a supervised machine learning task that entails classifying grayscale images of handwritten digits. The dataset contains 60,000 training examples and 10,000 test examples that have been size-normalized and centered in a 28×28 image. This multi-class classification problem is a basic yet classic task in machine learning, and in this report, it will be explored with five methods: k-nearest-neighbours (KNN), logistic regression, support vector machine (SVM), multi-layer perceptron (MLP), and convolutional neural network (CNN).

2 Methods

2.1 KNN

In an attempt to make the classifier generalize better, I tried preprocessing with PCA: with 100 principal components, it already explained about 90% of the variance and the error did decrease; but FSix's[1] deskewing technique lowered the error even more as the 1's are probably nearly perfectly classified after deskewing. Since doing PCA on the deskewed images doesn't improve the error by much, for all methods to be discussed later in the report, only deskewing is used as the preprocessing technique.
Since KNN model complexity decreases with increasing `k`, the number of neighbours to consider, my cross-validation stops when the validation error has been increasing for the last five values of `k`. Using this pruned 5-fold cross-validation and my implementation from the assignment, the lowest validation error was achieved by `k=3`, which means the model is probably becoming too simple after that. With `k=3` and Euclidean distance, KNN reached a test error of 1.79%.

2.2 linear regression

With the goal of minimizing the test error, I chose the multi-class approach to train all weights collaboratively instead of the one-vs-all approach of training binary classifiers independently. Specifically, I used the `SoftmaxClassifier` code from assignment with the bias term and L2-regularization added to allow higher fitting flexibility but also control overfitting, and made `lammy` (the strength of L2-regularization) and `alphaInit` (the initial learning rate of `findMin`) adjustable hyperparameters in addition to `maxEvals`, since the previous default value for `alphaInit` in `findMin` was too large to achieve convergence. After manually trying out several values for each of these hyperparameters on a subset of training examples, I got a sense of the range of values that works well with the MNIST dataset. However, to select the optimal combination of `lammy`, `alphaInit`, and `maxEvals`, I originally wrote my own cross-validation with the pruning condition that the validation error has to improve by a certain threshold to justify continuously increasing maxEvals, but it was slow compared to scikit-learn's parallelizable `GridSearchCV` that I learned to use later when optimizing MLP, albeit the same results. Using the same deskewing preprocessing discussed above and `GridSearchCV`, the best set of hyperparameter values that came out of 5-fold cross-validation on the whole training set is `lammy = 5, alphaInit = 0.001, maxEval = 100`; with these values, the test error is 4.41%.

2.3 SVM

I implemented multi-class SVM using vectorized $\sum_{c \neq y_i} \max\{0, 1 - w_{y_i}^T x_i + w_c^T x_i\}$ as the loss function and stochastic sub-gradient descent.[2] I was using `findMin` before with the piecewise gradients; it worked fine and the adaptive learning rate gave a lower error, but I just don't think the extra training time was worth the small decrease in error. With stochastic sub-gradient descent, however, there are more hyperparameters: `lammy` (to control overfitting with L2-regularization), `learning_rate` (to ensure convergence at a reasonable speed), `batch_size` (to balance convergence in the correct direction and speed), and `epochs` (to balance underfitting and overfitting). Using scikit-learn's `GridSearchCV` and taking into account the hyperparameter values from logistic regression as well as the relative sizes of the dataset and `batch_size`, 5-fold cross-validation gave `batch_size=1000, epochs=500, lammy=5, learning_rate=0.0001` as the optimal combination; the resulting test error was 4.91% (note this error is not deterministic due to random batches in stochastic sub-gradient descent).

2.4 MLP

As the neural network has significantly more parameters to train than the previous methods, I decided to use the SGD version of the one-layer MLP implementation from assignment with `alpha` (the learning rate of SGD) and `batch_size` as adjustable hyperparameters in addition to `epochs`, `hidden_layer_sizes`, and `lammy`. This is where I realized my own cross-validation is taking too long to complete and switched to scikit-learn's `GridSearchCV` to take advantage of running the job in parallel using all the processors. To be compatible with `GridSearchCV`, I made the classifier inherit from scikit-learn's `BaseEstimator` and `ClassifierMixin`, and implmented the `predict_proba` method. However, with `GridSearchCV`, it tends to select higher number of `epochs`, bigger `batch_size` and lower `learning_rate` as this will almost always decrease the validation error in a shallow network at the expense of longer running time; the trade-off between accuracy and fitting speed is not calibrated with `GridSearchCV`, it's nonetheless still faster than my own implementation so I decided to go with it. The set of hyperparameter values that came out of 5-fold cross-validation is `alpha = 0.001, batch_size = 700, epochs = 10000, hidden_layer_sizes = [200], lammy = 0.001`, which gave a test error of 1.56% (note this error also varies a little with random batches).

2.5 CNN

I followed Escontrela's[3] implementation but modified the structure to arrange other methods under `fit` and `predict` methods. The CNN's architecture is composed of two consecutive convolutional layers followed by a max pooling layer, then finally feed the flattened output to a MLP for classification. Necessary adjustments were made in code to allow varying hyperparameter values for `n1_filters` (number of filters in the first convolutional layer), `n2_filters`, `filter_size` (the same size was used for both layers), `conv_stride`, `mlp_size`, `batch_size`, `epochs`, and `learning_rate`. Since there are many combinations of these hyperparameters and that one run of CNN is not fast, I decided to go with scikit-learn's `RandomSearchCV` of 50 models instead. In addition to Escontrela's default values, I chose candidate hyperparameter values with Aszemi and Dominic's[4] work and some intuition like the relative size of the image and `filter_size`, dataset and `batch_size`. The set of hyperparameter values that came out of 2-fold cross-validation is `n1_filters = 8, n2_filters=5, filter_size=8, conv_stride=1, mlp_size=128, batch_size=64, epochs=2, learning_rate=0.01`, which achieved 1.68% test error.

3 Results

Model	Their Error	Your Error (%)
KNN	0.52	1.79
linear regression	7.6	4.41
SVM	0.56	4.91
MLP	0.35	1.56
CNN	0.23	1.68

4 Discussion

The reported test errors are generally close to the provided errors. With KNN, the provided error of 0.52% was achieved with shiftable edges preprocessing and non-linear deformation.[5] I think the 1.27% further decrease in error rate can be attributed to the latter, which captures non-linear local distortions so that the classifier is more robust to minor transformations like rotation and shifting on parts of the image. Even though the deskewing preprocessing that I use captures some rotation invariance on the whole image, I think it is also valuable to capture local distortions. Moreover, a different distance function, for example, Tangent distance and Mahalanobis distance, may be more suitable in this case than the Euclidean distance.

My error for linear regression is 3.19% lower than the provided error because their error was achieved with pairwise linear classifier[6] while mine was a multi-class logistic regression, which calibrates all weights together, instead of fitting them independently, so that the correct class has the highest probability. Further improvement may be made through more thorough hyperparameter tuning and trying different regularization schemes, but I think this is near the limit of what *linear* classifiers can do.

My error for SVM is 4.35% higher than the provided error because of their use of virtual SVM,[7] which incorporates known invariances of the problem by generating artificial training examples to include in the set of support vectors found by the first run of SVM. Their preprocessing included translating the input images by 2 pixels horizontally or vertically (but not both), in addition to deskewing them. Their implementation incorporates more invariance hence more accurate support vectors than mine does, and a pitfall of my multi-class SVM is that it assumes a linearly separable data in input space. To improve, I think data augmentation can be used to generate horizontally and/or vertically translated training examples like they did, kernel SVM can be used to loosen the linearly separable assumption, and an adaptive learning rate may also help.

The 0.35% error rate of their 6-layer MLP[8] (12.11 millions of weights) comes at the expense of heavy computation. One interesting implementation detail is that the entire training set gets deformed by elastic distortion, rotation, scaling, and horizontal shearing at the beginning of every epoch, so that the network is insensitive to in-class variability. My implementation of MLP has only 1 hidden layer. An obvious choice to improve is to increase the depth of the network; with increasing depth, dropout should be considered in addition to L1- or L2-regularization, other activation functions and optimizers should be added to the adjustable hyperparameters to avoid vanishing gradients and escape local minimum.

Their CNN achieved an impressive error of 0.23% with an ensemble bagging of CNNs called multi-column deep CNN,[9] where an arbitrary number of CNNs are trained on inputs preprocessed (e.g. width-normalized) and distorted in different ways and the final predictions are obtained by averaging individual predictions of each CNN. No wonder their error is lower than mine! Since such an ensemble method can be very resource-intensive, other ways to improve the test error can include trying out different CNN architectures, using padding to retain more information at the borders, adding dropout, L1- or L2-regularization, and using data augmentation and different activation functions as the depth increases.

References

[1] FSix. *Introductory exploration of MNIST: Deskewing.*
https://fsix.github.io/mnist/Deskewing.html

[2] Vivek Srikumar. *SVMs: Training with Stochastic Gradient Descent.*
https://svivek.com/teaching/lectures/slides/svm/svm-sgd.pdf

[3] Alejandro Escontrela. *Convolutional Neural Networks from the ground up.*
https://towardsdatascience.com/convolutional-neural-networks-from-the-ground-up-c67bb41454e1

[4] Nurshazlyn Mohd Aszemi, P.D.D Dominic. *Hyperparameter Optimization in Convolutional Neural Network using Genetic Algorithms.*
https://thesai.org/Downloads/Volume10No6/Paper_38-Hyperparameter_Optimization_in_Convolutional_Neural_Network.pdf

[5] Daniel Keysers, Thomas Deselaers, Student Member, IEEE, Christian Gollan, and Hermann Ney, Member, IEEE. *Deformation Models for Image Recognition, 2007.*
http://www.keysers.net/daniel/files/Keysers--Deformation-Models--TPAMI2007.pdf

[6] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. *Gradient-Based Learning Applied to Document Recognition, 1998.*
http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf

[7] Dennis Decoste, Bernhard Scholkopf. *Training Invariant Support Vector Machines, 2002.*
https://people.eecs.berkeley.edu/~malik/cs294/decoste-scholkopf.pdf

[8] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, Juergen Schmidhuber. *Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition, 2010.*
https://arxiv.org/abs/1003.0358

[9] Dan Cirean, Ueli Meier, Juergen Schmidhuber. *Multi-column Deep Neural Networks for Image Classification, 2012.*
https://arxiv.org/abs/1202.2745