MNIST Handwritten Digit Classification

Jackie Liu, jl2627

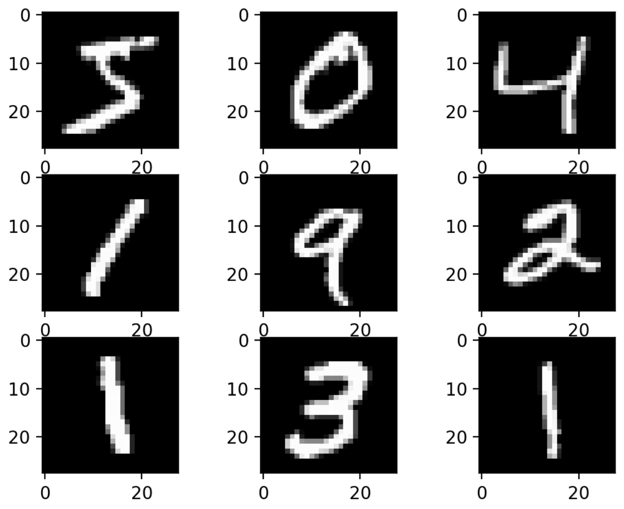
CS3220 Final Project

1 Introduction

A classic introductory problem in computer vision is that of handwritten digit classification. Given a matrix of pixels (i.e. an image) encoding a handwritten digit, our task is to construct a program to classify it as the digits from 0 to 9. A human can instantly understand the semantics given an image of a handwritten digit, but it is quite difficult to hardcode features of digits and then algorithmically classify them. Such an approach can be very brittle against variations in handwriting, and is too arduous for general image classification. The data-driven approach is to have general models which are parameterized to perform some specific prediction task based on what data they are trained on. These parameters can be computed through algorithms that solve a cost minimization problem with instances of labelled data. In this document, I explore common models used in terms of key properties such as prediction performance and accuracy. I followed a machine learning library's documentation and course notes for image recognition[1].

1.1 Data format

The data was downloaded from online[2]. The data from this source is the original MNIST dataset for this problem that has been modernized into csv files without extraneous information about offsets, etc. There are two files: data.csv and test.csv. The former contains 60,0000 row vectors of flattened 28×28 images of greyscale handwritten digits and the first column contains the label 0-9. The latter contains 10,000 different images to evaluate the models against (the test digits contain labels to check against). It is a best practice to partition data into an additional validation set for hyperparameter tuning, so in some cases we will be using such a validation set. In image form, the vectors look like the following:



[1]https://scikit-learn.org/stable/index.html & https://cs231n.github.io/

[2]https://pjreddie.com/projects/mnist-in-csv/

k	time to train (s)	time to predict (s)	error rate (%)
5	26.1	1003.6	3.12
50	26.2	997.0	4.66

Fig. 1: Performance benchmarks of k-nearest neighbors, $n = 50,000$

k	time to train (s)	time to predict (s)	error rate (%)
5	0.031	9.29	26.03
3	0.031	9.18	25.63

Fig. 2: Performance benchmarks of k-nearest neighbors, $n = 500$

1.2 Notation

We have $n_r = 60,000$ and $n_e = 10,000$. $d^{(i)} \in \mathbb{R}^{784}$ (the images have integer grayscale values from 0 to 255, to be more precise) is the ith training image vector with label $l^{(i)} \in \{0, 1, ..., 9\}$. Also for test $t^{(j)} \in \mathbb{R}^{784}$, with labels $q^{(i)}$.

1.3 Measuring performance

The models will be trained with data.csv and evaluated on test.csv by the metric of percent misclassifications (e.g. classifying a 0 as a 9). Although a lower rate of misclassification is optimal, other considerations include training time and prediction speed.

1.4 Technology

I used Google hosted Colab notebooks with the python3 runtime to execute code interactively. The machine learning packages I used were scikit-learn. To run the code with no setup, use a similar hosted notebook program that can run bash scripts. Mount the provided bash script or copy and paste it into a cell and run it to download the data.

2 A simple model: K-Nearest Neighbors

This approach does no training, only remembers all $d^{(i)}$ in linear time and linear space. Time to train is relatively quick. However, prediction complexity suffers, as it takes linear time: for a given digit to predict $p \in \mathbb{R}^{784}$, it scans all $d^{(i)}$ and computes $||d^{(i)} - p||$ and takes the mode of the corresponding $l^{(j)}$ of k vectors $d^{(j)}$ such that $||d^{(j)} - p||$ is minimal for some distance metric (eg L1 or L2 norms of images as row vectors is natural).

2.1 Results

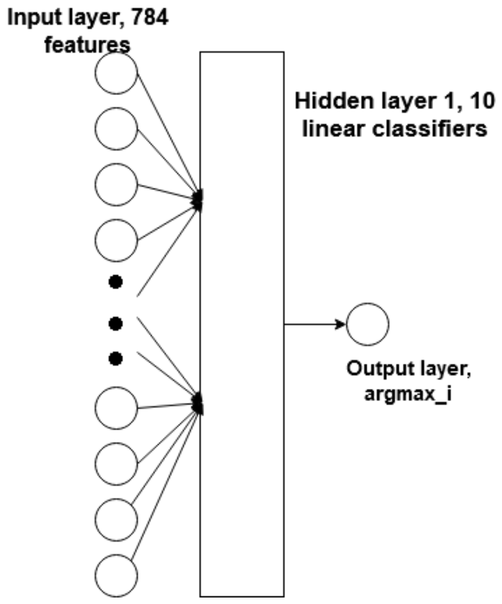
Using the sklearn kNN implementation with the L2 norm and $k = 5$ and $k = 50$, I got sub 5% errors. This classification performance is comparable to the reference in LeCun et al. 1998 and Kenneth Wilder, U. Chicago[3] for L2/L3 kNN classification with no preprocessing, so further hyperparameter tuning does not seem to really lead to much better results short of performing some preprocessing. While classification accuracy is acceptable for such a simple technique, one of the biggest bottlenecks, as expected, is the extremely lengthy prediction time (roughly a tenth of a second per digit). We see that the better k in this case is $k = 5$, and prediction time is roughly the same, which is slightly unexpected as taking the mode of the closest 50 should take more time than 5.

If we reduce the training set size by two orders of magnitude to $n = 500$ to have a more manageable prediction time, as expected the time to train and predict is reduced by two orders of magnitude. However, the error rate is perhaps unacceptable. Thus, a better approach is needed.

[3]scroll down to the table in http://yann.lecun.com/exdb/mnist/

3 Towards neural networks: multinomial logistic regression

The technique of k-nearest neighbors captures only the notion of a geometric area as the meaning of a given digit. In the large neural networks used in deep learning, it is believed that the many layers of neurons learn properties of digits as humans would. In addition to better prediction accuracy and faster prediction times, large neural network based approaches are favored over k-nearest neighbors despite its simplicity. A basic building block of larger neural networks is the linear classifier, and we will use such a linear classifier called multinomial logistic regression in ten neurons of a "neural network". That is, we have a single layer neural network with 10 linear classifiers as neurons. That is, we have the following architecture, with an edge from every feature node to every linear classifier:


Input layer, 784 features
Hidden layer 1, 10 linear classifiers
Output layer, argmax_i

The network takes a digit x and uses its learned weights $w_i \in \mathbb{R}^{784}$ (and an optional bias $b \in \mathbb{R}^{10}$) to output a scores vector $\hat{y} \in \mathbb{R}^{10}$ where each

$$\hat{y}_i = \frac{e^{w_i^T x}}{\sum_{j=0}^{9} e^{w_j^T x}}$$

corresponds to the confidence that x is the digit i. More formally,

$$\hat{y}_k = \mathbf{Pr}[Y = k \mid X = x; W]$$

a probability distribution over the digit classes parameterized by the weights. The "linear" part comes from the inner products, which are fed as input to the softmax function "activation" part. The final prediction is $\mathbf{argmax}_i \ \hat{y}_i$. To learn the weights w_i, we obtain from maximum likelihood estimation the so-called cross-entropy loss function

$$\mathcal{L}_i(d^{(i)}, l^{(i)}, W) = -[\mathbf{log}\ \hat{y_k}]$$

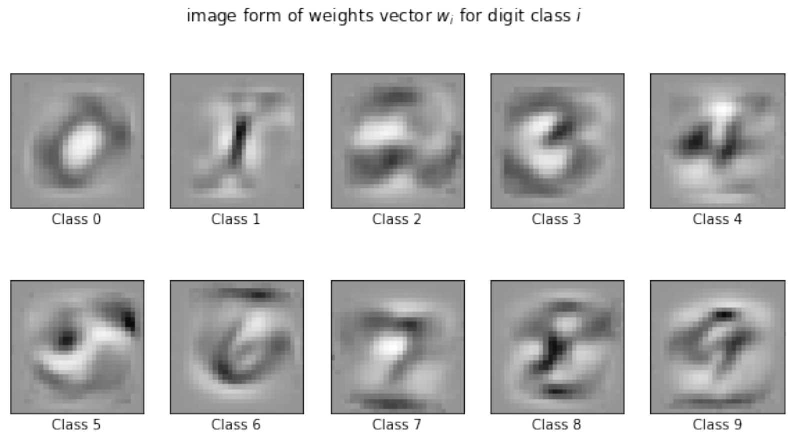
where k depends on $l^{(i)}$. Then, over all training examples we have the total loss

$$\mathcal{L} = \frac{1}{n} \sum_i \mathcal{L}_i$$

We can use gradient descent to minimize the loss function and find the weight parameters by deriving the gradient of $\mathcal{L}$ with respect to our weights. Specifically, we would use a more efficient formulation of gradient descent, such as stochastic gradient descent (e.g. a sample of $n = 64$ training examples) to approximate the full gradient and reduce the amount of sub-gradients we need to compute, since the training set is relatively large.

3.1 Results

Using the corresponding implementation in scikit-learn, there was 10.03% average error rate (slightly better than the 12% of LeCun et al.) and a significant improvement of 0.027 seconds average prediction time (over 100 trials), with only a 13.71 seconds training time. Although the error rate almost tripled, the prediction time is orders of magnitude better, with training taking half the time. We can visualize the 10 learned weight vectors in each neuron by turning them into images:



We see that the model learned a representation of each of the digit classes, and the act of taking inner products "matches" an input digit with each representation, and each representation looks reasonably like a template for each digit class.

4 Non-linear decision boundaries: adding a layer

The massive performance gains of the multinomial logistic regression approach motivates further experimentation. Let's see how we can improve the single layer neural network with linear classifiers. Linear models, like k-nearest neighbors, draw decision boundaries for image vectors in high dimensions. However, taking inner products results in decision boundaries that are linear, and in general decision boundaries may not be linear. In 2 dimensions for example, we could have a decision boundary between two classes that is actually a quadratic function. In addition, we learn only one template for each digit class corresponding to the ten weight vectors w_i.

One extension would be to add an additional layer that takes the output of the ten linear classifiers as input to another ten linear classifiers. If we collect the weight parameters into a weight matrix, then there are now two weight matrices $W_1 \in \mathbb{R}^{10\times 784}$ and $W_2 \in \mathbb{R}^{10\times 10}$ in the first and second layer, respectively. An input digit x would travel through the network before reaching the output layer as

$$\sigma(W_2\mathbf{max}(0, W_1x))$$

where $\mathbf{max}(0, b)$ is the "Rectified linear unit" activation function between the hidden layers and σ is the softmax function. The ReLu function is essentially to introduce non-linearity between the linear units and avoid issues such as a vanishing gradient. We would use the same cross-entropy loss function, but now we would need to use the technique of back-propagation to compute the gradients for gradient descent.

4.1 Results

Training time was long at 137 seconds, with an average error rate of 6.4% and average time of 0.033 seconds. Accuracy nearly doubled (approaching k-nearest neighbors), while prediction time stayed the same. While training times get long, in exchange we get improved accuracy and prediction speed from k-nearest neighbors. Interestingly, the time complexity on neuron count was $O(h^k)$ where h is the number of neurons per hidden layer and k is the number of hidden layers, but this time complexity trend becomes sublinear with a 500×300

HL tuple	time to train (s)	time to predict (s)	error rate (%)
(10, 10)	137	0.033	6.4
(500, 300)	413	0.4	2.3

Fig. 3: Performance benchmarks of neural nets with linear classifiers

network, and time to train only increased by a seeming constant factor. Nevertheless, we exceed k-nearest neighbors performance and have comparable accuracy in the many iterations of 2 and 3 layer nets in Hinton and LeCun et al. Time to predict increased by an order of magnitude, but it is still negligible. We begin to see the hunger for scale in neural networks as the number of layers and neurons increases for better performance.

5 Appendix

5.1 Regularization

In practice, to penalize overfitting when learning weights, a regularization term $\alpha R(W)$ is added to the loss $\mathcal{L}_i$. The regularization term adds to loss by some metric of model complexity. In my experiments, I used L2 regularization, which in simplest form is the squared Frobenius norm of the weights matrix: $R(W) = \sum_i \sum_j W_{i,j}^2$. α was usually at a default of 0.0001. Larger α increase overfitting penalty and decreases variance while smaller α decrease bias (underfitting) by encouraging more model complexity.

6 Conclusions

We have explored how to accomplish the task of predicting handwritten digits through data-driven means and achieved a fast, accurate predictor with a moderately sized neural network. With the tools of linear algebra, probability, and optimization, we have turned the task of recognizing digits from one based on visual cognition to minima finding and vectorized operations. While the performance of our fully connected network of linear classifiers is near state of the art for this problem, we could see even better performance with a hierarchical architecture through the technique of convolution, and further explore more general image classification tasks with the same techniques.