Comparing Results of K-Nearest Neighbor and Feedforward Neural Network on Character Recognition

Yifan Wu
Undergraduate Student
Vanderbilt Univeristy School of Engineering
Nashville, TN, USA
Yifan.Wu@Vanderbilt.edu

***Abstract*—This paper experiments with two of the basic machine learning algorithms, k-nearest neighbors and feedforward neural network using Stochastic gradient descend, and compared their results on a self-generated character recognition dataset.**

Keywords—Optical Character Recognition, K-Nearest Neighbors, Stochastic Gradient Descend, Feedforward Neural Network

I. Background Information

It might be surprising to see I jumped from the topic of business card recognition and classification to this basic algorithmic design and comparison. However, if I were to identify one thing I learned from EECE 4353 and EECE 3891-01, it would be that building everything from scratch helps with the understanding of the material a lot. This has been my idea going into this assignment from the beginning – instead of choosing something like facial recognition which I am certain I am uncapable of doing, I picked a simpler project in computer vision, which is business card recognition. However, as I go deeper into the subject, I found there are still many layers of knowledge building on top of each other, each worth a dedicated project to. To name a few, recognizing bounding boxes and grouping related boxes together, binarizing the image adaptively based on the surroundings, performing meaningful feature extraction before processing, etc. I found it more and more unfeasible to build everything from scratch. Therefore, instead of using an outside library such as MATLAB's (which is a customized implementation of the Tesseract OCR algorithm) and risking not understanding the material fully, I chose to scale down the project again to something I can feel confident explaining every detail afterwards.

This project focuses on first constructing a dataset of letters, then train two different machine learning algorithms on it, and compare the results and the process. Most of the time on this project are dedicated to understanding Stochastic gradient descend, which I mostly get reference from [1]. The video by 3Blue1Brown [2] also helped a lot on explaining the concept and implementation in a feedforward network visually.

An interesting read related closely to business card recognition is [3], which classifies whether an image is a promotion or not. They take a crude approach towards both the OCR and classification part, but the result ended up accurate. To identify which stage of processing requires refined result and which are better left rough may be something I need to build intuition for in the future to better manage time and resource in a project.

II. Generating Dataset

The first step to all machine learning problem is to find a good, well labeled dataset. While there exist many well documented datasets, namely the MNIST dataset of handwritten digits [4], it is also important to explore the possibility of generating a dataset on demand. There are many advantages that comes from generating a dataset:

- Not be limited by dataset size
- Able to recreate similar noise and environment as the testing data of interest
- Generated data contain labels by default
- Able to modify training data if the goal of the project changes
- Know for sure the encoding method of the dataset

Therefore, in this project I will attempt to generate my own dataset.

Since most OCR algorithms functions on unit of characters, it is most natural to begin with generating a set of characters. I narrowed my range to the alphabet, both capital and lowercase, and numbers 0 through 9. These characters are the most used and generally have enough detail to be different from any possible noise, unlike "," and "." where they can be easily treated as noise given a noisy image.

I experimented with various sizes of image sizes, ranging from 10 by 10 pixels to 200 by 200, and eventually landed on two promising ones, 15 by 15 and 28 by 28. I noticed that there is not much detail needed to recognize the characters, and the smaller the image is, the faster the algorithm will run. I took

inspiration from the MNIST dataset [4], which each digit is 28 by 28 pixels. Having this popular dataset as reference later become important for finding resource on the common parameters for tuning algorithms.

Then, I picked some common fonts including Arial, Calibri, Times New Roman, and many more. It is important to make sure no symbolic fonts or stylistic fonts exists in the set. Although it is tempting to simply use MATLAB's **`listfonts`** function, I noticed it is often too inconsistent to include all system fonts. Eventually, I hand picked about 30 fonts for the training dataset and 20 for the testing dataset. It is important to use different fonts for the training and testing, since I will be centering the characters later, having the same character from the same font will be too easy for the algorithm.

I used MATLAB's **`insertText`** function to insert the character into the image, randomly picking the font it uses and the character it puts in. Then, I randomly rotate the image by with in ±15 degrees, so that if two identical characters were ever chosen, their pixel values will be different. The small rotation angle also maintains the upright-ness of the characters, keeping a "6" from being confused with a "9".

Since I will be using the dataset myself, I can preprocess the dataset and make it easier to access. That is, to flatten each image to a row vector, and layer all the data on top of each other. This way, I can skip reading each image file in the training stage.

Fig. 1. Example of a generated character "3" rotated slightly clockwise.

Like mentioned before, self-generated dataset can also be applied with similar noise as image of interest. For example, **Figure 2.** is a character from the dataset originally generated for the business card recognition project. Where the letter has been applied similar process which a business card may go through before passed to the OCR algorithm, including Canny edge detection, gray-scale morphological reconstruction and binerization.

Fig. 2. Example of a generated character "w" applied with Gaussian noise, Canny edge detection, Gray-scale morphological reconstruction, and binerization.

For this project, I generated a set of 5000 training characters and 1000 test characters of size 28 by 28 pixels.

III. K-Nearest Neighbor Classification

The k-nearest neighbor algorithm brings the data into a n-dimensional space where n is the number of feature data have. For this project, since we are simply flattening the images without sophisticated feature extraction, each pixel will be a feature, resulting in 28×28=784 features.

K-nearest neighbor is a lazy learning algorithm, meaning it lacks a distinct training phase unlike most neural networks. It works by first loading all the training data into the 784-dimensional space, then put in the testing data and calculate the Euclidean distance between the testing data and all other training data. Then, it will select the k-nearest training data to the testing data and look at each of their label. Finally, the testing data will be classified as the majority label of the group. This process can become exponentially slower as the training dataset increase in size and each image increase in number of features.

I tested values for k as a hyperparameter and eventually concluded the ideal k for this dataset is 3. With more data in the dataset, a higher k gives more reliable results, while in a smaller dataset, or one with similar elements such as lowercase L, "l", and number one, "1", using a small k and go with the algorithm's first instinct may give better result.

The k-nearest-neighbors got an accuracy of 71.4%. Since the dataset contains both capital and lowercase letters, and there are a lot of characters looking similar, this result is better than it looks at first glance.

IV. Feedforward Neural Network with Back propagation

The feedforward neural network is the focus of this project. We have discussed and implemented a Bayes Regression in EECE 3891-01, where a previous distribution combined with a likelihood function can generate a posterior distribution. However, how neural networks updates the weights and bias were not apparent to me. So, I take this as an opportunity to explore.

A. Constructing Network

The first step is to construct a neural network we can work with. I created a function called **`fnn`** where it can generate the appropriate weight and bias matrices provided the number of inputs, number of hidden layers, number of nodes in each layer, and number of outputs. All weights and biases are initialized to a random number uniformly distributed between -1 and 1.

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

Fig. 3. Example of a weight and bias matrix with input $x_{1,2,3}$, output $y_{1,2}$

Since the weight and bias matrix dimensions between layer can be different, I isolated the first and last layer, stored them as separate variables, and had all the layers in between as a cell array, each containing the appropriate weight and bias matrices. These weight matrices also need to be passed around often, thus I often group them into a single *Weights* cell array.

B. Stochastic Gradient Descend

The Stochastic gradient descend is an iterative method for optimizing a function. It is the Stochastic estimation of the gradient descend. Where instead of using the whole training dataset and finding the best possible direction and step to take at each location, the idea of batch and η, the learning rate, is introduced, where we only look at a very small subset of the training dataset at a time (*i.e.* 32 or 64 out of 5000) and take the step after each batch. This allows us to take rapid, less calculated, steps rather than slow, well calculated steps. Although sometimes we may take a step in a nonoptimal direction, the cost for calculating that step and the step to recover are often trivial. The equation I used to update the weights is:

$$w = w - \frac{\eta}{\mathrm{n}} \sum_{\mathrm{i}=0}^{n} \nabla \mathrm{Q}_{\mathrm{i}}(\mathrm{w})$$

Fig. 4. The equation for updating weights using Stochastic gradient descend. w is the weight, η is the learning rate, n is the batch size, Q_i is the value of the loss function of the i-th datum.

Both learning rate and batch size are hyperparameters, which means they are generally tuned through testing, and weight is generated randomly at first. Therefore, the only thing left to calculate in the equation is ∇Q, the loss, or, the amount of change suggested.

C. Back Propagration

In order to determine the appropriate loss for each weight and bias, we can use the method called back propagation.

The output of the neural network is represented in one hot encoding, meaning the correct label has a value of one while all other labels have a value of zero. Since this is a supervised training, meaning the correct label are given to the training dataset, we can use a cost function to calculate the difference between the estimated result and the true label. Here, we used the sum of the square of the difference illustrated by **Figure 5.**

$$\mathrm{C_i} = \sum_{\mathrm{j}=0}^{\mathrm{m}} \left(\mathrm{a_j} - \mathrm{y_j}\right)^2$$

Fig. 5. Cost for the i-th datum is the sum of the square of the difference with the expected output. j is the index of the output node, m is the number of output nodes, and a is the value of a node after an activation function, or simply, the predicted value of a node.

Having the cost, we can properly evaluate the performance of the network. However, cost is just a single number, what we are interested is how changing each weight and bias can affect the cost. (**Figure 6.**)

$$\frac{d\mathrm{C}}{d\mathrm{w_L}} \text{ and } \frac{d\mathrm{C}}{d\mathrm{b_L}}$$

Fig. 6. C represents the cost for one datum, since weights and biases uses subscripts to represent the layer they are on, in this case L, meaning the last layer, I dropped the subscript of C to avoid confusion.

We have three important equations here that can help us propagate back. (**Figure 7.**)

$$C = (a_L - \mathrm{y})^2$$
$$z_L = w_L a_{L-1} + b_L$$
$$\mathrm{a}_L = \sigma(\mathrm{z}_L)$$

Fig. 7. C is the cost as before, and z is a shorthand representation of the value of a node before going through the activation function.

Here, we use *z* for a shorthand notation of the value of a node before going through the activation function, and *a* is the value after activation function. We are not going in depth into activation functions in this project, so we simply picked the classic sigmoid function. (**Figure 8.**)

$$\sigma(\mathrm{x}) = \frac{1}{1 + \mathrm{e}^{-\mathrm{x}}}$$

Fig. 8. The sigmoid function squeshes all values to between 0 and 1.

We are interested in the change of cost with respect to the change of weight and bias. Since we have the equations now, it is only a simple chain rule until we get the result. (**Figure 9.**)

$$\frac{d\mathrm{C}}{d\mathrm{w_L}} = \frac{d\mathrm{z_L}}{d\mathrm{w_L}} \frac{d\mathrm{a_L}}{d\mathrm{z_L}} \frac{d\mathrm{C}}{d\mathrm{a_L}}$$
$$\frac{d\mathrm{C}}{d\mathrm{b_L}} = \frac{d\mathrm{z_L}}{d\mathrm{b_L}} \frac{d\mathrm{a_L}}{d\mathrm{z_L}} \frac{d\mathrm{C}}{d\mathrm{a_L}}$$

Fig. 9. The chain rule expansion of the derivative.

Then, simply take the derivative of the respecting terms.

$$\frac{d\mathrm{C}}{d\mathrm{w_L}} = a_{L\text{-}1} \cdot \sigma'(\mathrm{z}_L) \cdot 2(a_L - \mathrm{y})$$
$$\frac{d\mathrm{C}}{d\mathrm{b_L}} = \sigma'(\mathrm{z}_L) \cdot 2(a_L - \mathrm{y})$$

Fig. 10. Calculating the derivatives.

By applying this equation recursively with decreasing layer index, we will eventually calculate all the ideal changes to all weight and biases in the network, and this is ∇Q_i. However, remember this is only how one datum in the training dataset want the weight to change. We need to calculate this for all data in a batch and then we can update the weight and bias once. After applying this algorithm for all batches, we have completed one *epoch*, that is, one use of all training data in the dataset. However, since we broke the data into batches, we can reuse the data again with a different permutation of batches.

D. Hyperparameter Tunning

Tunning hyperparameters for feedforward neural network is more difficult compared to k-nearest neighbor where there is only one k to tune. In feedforward neural networks, I experimented with different batch sizes ranging from 16 to 1000, learning rate from 0.01 to 0.9, number of layers from 1 to 10, and nodes per layer from 16 to 100. Eventually, I found the combination of batch size 32, learning rate 0.9, 2 hidden layers with 64 nodes each to be the best combination for my own dataset. However, this is far from perfect since it only achieved an accuracy of 55.6% (**Figure 11.**), far from the ideal range

which is around 96% and even the k-nearest neighbor implementation. However, this still came a long way from training 3000 epochs only end up with 22.4% accuracy. This is generally thanks to me having a similar dataset as the MNIST handwritten digits, so I can find inspiration from the state-of-the-art MNIST network and test parameter values similar to the ones used there.

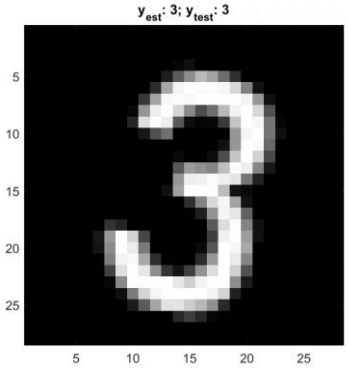

Fig. 11. Feedforward estimation of image. Estimate is 3, true value is 3.

V. Comparing Results From K-Nearest Neighbor and Feedforward Neural Network

In terms of simplicity, KNN is easier to build. Simply calculate Euclidean distance of the test to all training data and sort them based on distance will do most of the trick. Tuning KNN is also easy, a simply for-loop over a range of k and recording the best accuracy is the way to go. Whereas for FNN, weight and bias calculation involves a lot of matrix multiplication, and since each layer of weight and bias are of different dimensions, keeping the code organized and clear is hard. Moreover, tuning FNN is much harder. It requires optimizing four hyperparameters, which is often impossible to do without background knowledge of the conventions.

In terms of evaluation time, KNN's evaluation time exponentially increase as input dimension and training dataset increases. However, with small input dimension, such as the 15 by 15 pixels images, KNN is very fast in providing a result. FNN generally takes a longer time in the training phase. However, after trained, the network evaluates test input almost instantly because it is just a series of matrix multiplication.

In terms of understandability, KNN makes much more sense. However, it also relies heavily on preprocessing the data. For example, given images of t-shirts and hoodies, it may be hard to classify. However, if we were able to extract the length of the sleeves and if it has a hood in the preprocessing stage and use these as the features, KNN will see much better results. As for FNN, it can for the most part be treated as a black box. With enough tuning and training time, it can almost perform well with any data provided.

VI. Next Step

I learned a lot about KNN and FNN through this project, and I find building from scratch is indeed a good way to learn. For next steps, since I have experience with the most basic form of neural network, I would like to experiment with RNN, CNN, or even LSTM and see what they are doing differently to make them perform better for particular tasks.

I would also like to continue my project with business card recognition. That is, to experiment with performing OCR on image with noise or distorted. I would also like to learn how a bounding box finder works and possibly build one myself.

VII. Conclusion

In this project, I built a k-nearest neighbor algorithm and a feedforward neural network with SGD from scratch and tested them against the dataset I generated on my own. Neither of the algorithms performed particularly well on the dataset, with accuracy of 71.4% and 55.6% from KNN and FNN respectively. I am unsure if this is with my tuning or from the dataset itself. Compared to the MNIST handwritten digits, this contains more than six times the possible outputs and a lot more similar characters that are even impossible for humans to tell apart. Since this is ultimately a learning experience, I would call this project a great success. Source code I wrote for this project can be found under the GitHub repository: https://github.com/WilliamW9758/OCRwithKNNandFNN

References

[1] J. Kiefer and J. Wolfowitz, "Stochastic estimation of the maximum of a regression function," *The Annals of Mathematical Statistics*, vol. 23, no. 3, pp. 462–466, 1952.

[2] 3Blue1Brown, Backpropagation calculus | Chapter 4, Deep learning. YouTube, 2017.

[3] Hubert, P. Phoenix, R. Sudaryono, and D. Suhartono, "Classifying promotion images using optical character recognition and naïve Bayes classifier," *Procedia Computer Science*, vol. 179, pp. 498–506, 2021.

[4] Y. LeCun, "The mnist database," *MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges*, 1998. [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed: 18-Dec-2021].