Deep Learning Architectures for Hard Character Classification

Vy Bui¹, Lin-Ching Chang¹

¹Department of Electrical Engineering and Computer Science, The Catholic University of America, Washington, District of Columbia, USA
01bui@cua.edu, changl@cua.edu

Abstract – Recent research indicates that deep learning has achieved noticeably promising results in a wide range of areas such as computer vision, speech recognition and natural language processing. This paper offers an empirical study on the use of deep learning techniques for hard characters recognition on the notMNIST dataset. The MNIST dataset has been widely used for training and testing in the field of machine learning such as for the performance comparison of different deep learning algorithms. However, similar performance evaluation using the notMNIST dataset has not been reported. This dataset is harder and much less clean than the MNIST dataset. In this paper, we constructed several experiments to evaluate various deep learning architectures and proposed a multi-layer convolutional neural network for large-scale hard character classification on the notMNIST dataset. The result shows that our method can achieve 98% accuracy of classification. Comparisons were also performed against conventional fine tuning models such as logistic classifier and shallow neural network to demonstrate that well-constructed deep neural networks can significantly improve the accuracy of hard character classification on the notMNIST dataset.

Keywords: Deep learning, character recognition, neural network, convolutional neural network, image classification

1 Introduction

Recently, learning through deep models draws significant attention. Various deep learning architectures such as deep neural networks, deep convolutional neural networks, and recurrent neural networks have been applied to a wide range of fields like computer vision, automatic speech recognition, or natural language processing. The concept of deep learning is to discover the features in the input data in multiple levels of representation, in other words, higher level features represent more abstract semantics of the data. Deep learning techniques have been shown to produce state-of-the-art results on several difficult tasks [1-5]. For examples, computer vision tasks such as image classification based on visual content is a very challenging task, mostly because of the intra-class variation, scale variation, viewpoint variation, deformation or occlusion. Numerous efforts have also been made to recognize handwritten digits on the famous MNIST dataset [6] using conventional machine learning techniques like linear/nonlinear classifier, K-nearest neighbors, and SVMs, or deep learning architectures like deep neural networks and deep convolutional neural networks [1-3]. These studies have reported that the deep learning architectures outperform the conventional methods and are able to achieve near-optimal human performance.

In this paper, we constructed several experiments of large-scale character datasets drawing from the notMNIST dataset [7] to evaluate various deep learning architectures. We then proposed a five-layer deep convolutional neural network for character recognition on the notMNIST dataset. This classification task is more challenging comparing to the handwritten digits recognition on the MNIST dataset. We also compared the classification accuracy of the proposed architecture against several conventional fine tuning models such as logistic classifier and shallow neural networks.

The rest of the paper is organized as follows. In section 2, we provide the necessary theoretical background of neural networks. Section 3 describes the notMNIST dataset and the preprocessing steps used in this study. The detailed description of the conducted experiments, the proposed model, and the testing results are presented in section 4, followed by the conclusions in section 5.

2 Theoretical Background

Character recognition using different features has been extensively studied, and a nice survey on feature extraction methods for character recognition can be found in [8]. In this section, we provide the necessary theoretical background of a neural network that will help our readers to understand the empirical experiments we have performed. Two types of neural networks will be described: regular neural networks and convolutional neural networks. The specific architectures used in our experiments will be discussed in section 4.

2.1 Score Function

A score function maps the pixel values of an input image to a confident score for each class. A learning system computes a score function that can be defined in equation (1):

\[ Y_i = F(X_i, W, b) \]  

, where \( X_i \) \((i = 1, ..., N) \in \mathbb{R}^D \) is the \(i\)-th input image with \(D\) dimension and \(N\) is the total number of training images. \(W\) and \(b\) represent the collection of adjustable parameters in the system. The output \(Y_i \in 1 ... K\) is the recognized class label of a given image \(X_i\), or the score associated with each class, and \(K\) is the total number of distinct classes. For example, in notMNIST we have a training set of \(N = 500k\) images, the dimension of each image is \(D = 28 \times 28 \times 1 = 784\), and the

number if classes is $K = 10$ (A-J). A simplest score function can be represented as a linear classifier:

$$Y_i = F(X_i, W, b) = WX_i + b \quad (2)$$

The pixels of an input image $X_i$ is arranged as a single column vector of size $D \times 1$. $W$ is a weight matrix of size $D \times K$ and the bias vector $b$ of size $[K \times 1]$ which contains the adjustable parameters of the score function. $W$ and $b$ determine the behavior of the score function, and directly influence the output scores $Y_i$. For the notMNIST dataset, the size of $X_i$ is $[784 \times 1]$; $W$ is $[10 \times 784]$; $b$ and $Y_i$ is $[10 \times 1]$.

In our experiments, we implemented a popular linear classifier called Softmax [9]. The advantage of Softmax is that instead of treating the output $Y_i$ as un-calibrated scores for classes, it converts these scores to a vector of values between zero and one with the sum of the values equals one. Therefore, these scores can be interpreted as the probabilities for classes:

$$P(Y_i | X_i; W) = S(Y_i) = \frac{e^{Y_i}}{\sum_{j} e^{Y_j}} \quad (3)$$

### 2.2 Cross-Entropy Loss Function

In equation (3), $X_i$ and $Y_i$ are fixed, and we can adjust $W$ and $b$ to optimize an objective function. Therefore, the goal is to fine tune these parameters so that the output probabilities of $Y_i$ given $X_i$ best match the ground truth labels $L_i$. An error function measures the discrepancy (cross-entropy) between the correct output for an input pattern and the output produced by the system can be defined in equation (4).

$$E_i = D(L_i, S(Y_i)) = -\sum_{i} L_i \log(S_i) \quad (4)$$

, where $S_i$ is defined in eq. (3) and $L_i$ is the correct output (or true label), and $D$ is the discrepancy between the true label and the output score.

A loss function measures the average of errors $E_i$ over a set of labeled samples (i.e., training dataset $\{(X_1, L_1), ... (X_N, L_N)\}$ can then be defined as equation (5)

$$L = \frac{1}{N} \sum_{i=1}^{N} E_{i}^{train} \quad (5)$$

Therefore, the objective of the training task is to find the optimal values of $W$ and $b$ that minimize the loss function $L$.

It is known that the gap between the error rate on the test set $E_{test}$ and the error rate on the training set $E_{train}$ will drop with the increased number of training samples [1].

### 2.3 Optimizing Loss Function

#### 2.3.1 Gradient Descent

Optimization is the process of finding the set of parameters $W$ and $b$ that minimize the loss function. In other words, we are searching a moving direction in the weight space to improve the weight vector so that it will return a smaller loss. The best moving direction is related to the direction of the steepest descend or the gradient of the loss function with respect to the parameters. There are many ways of performing the optimization, but Gradient Descent (GD) is probably the most popular way to optimize the loss functions in neural networks. Here two gradient descent algorithms are briefly described: Simple GD and Stochastic GD.

The simplest minimization procedure is the simple gradient descent algorithm where $W$ is iteratively updated in the negative direction of the gradient as shown in equation (6):

$$W_k = W_{k-1} - \epsilon \frac{\partial E(W)}{\partial W} \quad (6)$$

, where $\epsilon$ is the step size (or so called the learning rate) that decides how fast along this direction in the next iteration. The complexity in evaluating the gradient is linearly related to the number of parameters in the system. In our case, we have 7840 parameters and therefore 7840 evaluations of the loss function are needed in each iteration to obtain the best set of $W$ and $b$. The computational complexity will get worse as the modern neural networks often have millions of parameters in training data. Apparently, this approach is not scalable.

To address this large-scale problem, one can estimate the loss by computing the gradient over a small subset (i.e. a mini-batch) of training data. This approach is called Stochastic Gradient Descent (SGD), which is also known as the on-line update. The size of the mini-batch is an adjustable parameter to be determined which is usually constrained by the capacity of hardware memory. Common mini-batch sizes range between 50 and 256 samples, but can vary for different applications.. Equation (6) can then be modified, so an approximated version of average gradient will be used in the next iteration.

$$W_k = W_{k-1} - \epsilon \frac{\partial E(W)}{\partial W} \quad (7)$$

This procedure makes the parameter vector fluctuates around an average trajectory, but usually converges considerably faster than simple gradient descent on a large training set [1].

#### 2.3.2 Gradient Back-Propagation

The gradient-based techniques are limited to linear systems; therefore the back-propagation algorithms are introduced to calculate the gradient in non-linear systems.

Let’s assume that a system can be built as a cascade of modules, each of which is a function $X_{k} = F_k(W_k, X_{k-1})$,
where $X_k$ is a vector representing the output of the module, $W_k$ is the vector of tunable parameters in the module, and $X_{k-1}$ is the previous module’s input vector. If the partial derivative of $E_i$ with respect to $X_k$ is known, then the partial derivatives of $E_i$ with respect to $W_k$ and $X_{k-1}$ can be computed using the backward recurrence:

$$\frac{\partial E_i}{\partial W_k} = \frac{\partial F}{\partial W_k}(W_k, X_{k-1}) \frac{\partial E_i}{\partial X_k} \quad (8)$$

$$\frac{\partial E_i}{\partial X_{k-1}} = \frac{\partial F}{\partial X}(W_k, X_{k-1}) \frac{\partial E_i}{\partial X_k} \quad (9)$$

where $\frac{\partial F}{\partial W_k}(W_k, X_{k-1})$ is the Jacobian of $F$ with respect to $W$ evaluated at $(W_k, X_{k-1})$, and $\frac{\partial F}{\partial X}(W_k, X_{k-1})$ is the Jacobian of $F$ with respect to $X$ evaluated at $(W_k, X_{k-1})$. Equation (8) computes some terms of the gradient of $E_i$, while equation (9) produces a backward recurrence. We can average the local gradients over the training set to obtain the full gradient.

2.4 Multi-Layer Neural Networks

To describe multi-layer neural networks, we begin with the simplest neural network—one which includes only a single neuron/node. This neuron is a computational unit that can take several inputs $X_1, X_2, \ldots, X_N$, and generates an output which is given by equation (10)

$$F(W^TX) = F \left( \sum_{i=1}^{N} W_iX_i + b \right) \quad (10)$$

where $F: \mathbb{R} \rightarrow \mathbb{R}$ is called the activation function. In this study, the popular Rectified Linear Unit (ReLU) [10] and the recent introduced Exponential Linear Unit (ELU) [11] were selected to be the activation function. The ReLU activation function can be defined by equation (11),

$$F(x) = \max(0, x) \quad (11)$$

The ELU activation function can be defined by equation (12),

$$F(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases} \quad (12)$$

A single node neural network which uses ReLU activation function is shown in Fig. 1. The ReLU can be replaced by the ELU activation function in some applications.

A multi-layer neural network often have more than one hidden layers followed by an output layer of neurons. Multiple layers of neurons with nonlinear activation function allow the network to learn nonlinear relationships between input and output vectors. An example of two-layer neural network is shown in Fig. 2 where ELU is used in the first layer and ReLU in the second layer.

2.5 Convolutional Neural Networks (CNNs)

A general architecture of a convolutional neural network (CNN) usually consists three core layers: Convolutional Layer, Pooling Layer, and Fully-Connected Layer. A neural network without the convolutional and pooling layers is known as the regular neural network as we have mentioned in section 2.4. These three layers are stacked to form a complete CNN architecture.

The convolutional layer is the core building block of a CNN. This layer comprises of a set of learnable filters. During the forward pass, each filter is convolved across the whole input volume to produce a 2D activation (or feature) map. In other words, features were extracted from the input volume and give to the next layer to extract more complex features. In the context of image recognition, features can be patterns in lower layers and object parts in higher layers. For example, the first convolutional layer can take a raw image as an input, then different neurons along the depth dimension may activate in presence of various oriented edged, or blobs of color.

The spatial extent of the connectivity along the depth axis is always equal to the depth of the input volume. For example, suppose that the size of the input volume is $[28\times28\times1]$, and the filter is $4\times4$ as shown in Fig. 3, then each neuron in the convolutional layer can have weights cover a $[4\times4\times1]$ region in the input volume, and for a total of 16 weights. Since notMNIST consists of grayscale images, thus the depth of our input volume is 1.
It is a common practice to periodically insert a pooling layer in between successive convolutional layers in a CNN. A pooling layer is used to progressively reduce the spatial size of the representation. It reduces the number of parameters and the computation time needed in training the network. In some applications, it is also used to control the possible overfitting. In this paper, the pooling layer functions independently on every slice along the depth of the input image. It reduces the dimension (width×height) of the input using the MAX operation. The MAX operation gives maximum activations over non-overlapping sub-regions of the input volume. For example, suppose the size of input volume is \([4 \times 4 \times 16]\), in max-pooling, a pooling unit outputs the maximum activation in the \(2 \times 2\) input region, as illustrated in Fig. 4. After pooling we have the output volume of size \([2 \times 2 \times 16]\). Notice that the volume depth is unaffected.

The third layer in a CNN is a fully-connected layer that is considered as a regular neural network that we have discussed in section 2.4.

### 2.6 Overfitting in Neural Networks

Overfitting is a major problem for deep neural networks especially for deep learning that has multi-layer and contains a large number of parameters. Regularization and Dropout have been efficiently used to prevent neural networks from overfitting in training. One of the most popular methods for regularization is the L2 regularization [12]. The idea is to penalize the squared magnitude of all parameters directly in the loss, i.e. for every weight \(W\) in the network, we add the term \(\frac{1}{2} \lambda W^2\) to the loss, where \(\lambda\) is the regularization strength.

Another recently introduced approach is the Dropout approach [13]. It randomly drops units from the neural network during training that prevents units from co-adapting. In other words, dropout is employed by keeping only a neuron active with a certain probability during the training; otherwise setting its value to zero.

### 3 Data and Preprocessing

#### 3.1 notMNIST

The notMNIST dataset is designed to be similar to the famous digits MNIST dataset, while appearing a bit more like real data since the data is a lot less clean compared to the MNIST dataset. The dataset is made of characters rendered in a diversity of fonts of 28×28 gray-scale images. The labels are 10 classes from ‘A’ through ‘J’. The original released training set has about 500k labeled images and the test set about 19k images. In this paper, we further split the original training set into two parts including a training set and a validation set for parameter tuning. Furthermore, it is necessary to balance the data across classes. In particular, the training set contains 52900 images and the testing set comprises about 1870 images for each class.

#### 3.2 Data Preprocessing

Normalization is a common practice in data pre-processing stage in which the data is transformed to a specific range such as between 0 and 1, or between \(-1\) and \(+1\). In this paper, image normalization is required since the signal intensity values of raw images. In particular, the data is normalized to have approximately zero mean and standard deviation about 0.5 using equation (13). After normalization, the intensity value of images will be ranged between \(-0.5\) and \(0.5\).

\[
x = \frac{\text{pixel}_\text{depth}}{2} - \frac{x}{\text{pixel}_\text{depth}}
\]

(13)

where \(x\) is the signal intensity value of a pixel in the input image, and the pixel depth, \(\text{pixel}_\text{depth}\), is 255 since the images are in 8-bit gray scaled format.

### 4 Experiments and Discussions

In this section, we provide the detailed description of six experiments we have performed. We showed how we utilize multi-layer and convolutional neural networks in hard character recognition. We implemented these architectures using Python 2.7 with Tensorflow 0.7 API [14]. The implemented models were trained and tested on Amazon Elastic Compute Cloud (Amazon EC2) [15]. The Amazon EC2 provides scalable computing capacity in the Amazon Web Services which is suitable and cost-effective for our tasks. We also utilized the cluster GPU instances available on Amazon EC2 to accelerate the computation time needed for training the networks.

As mentioned earlier, character recognition on the notMNIST dataset is a more challenging task than the written digits recognition on the MNIST dataset. Therefore, the conventional methods that work well on MNIST may not work well on notMNIST. We have tested the Logistic Regression classifier [9] using the tools supported by scikit-learn version 0.17 [16]. We trained on 200k training samples...
and received the accuracy of 85.11% on 10k test data. As we will show later, neural network architectures can achieve better performances on the same training and testing dataset. We also compare the predictive accuracy of the Multinomial Logistic Regression using Simple Gradient Descent [9]. As discussed in section 2.3.1, simple gradient descent is not scalable for a large dataset. The training time is irrational if we would load all the data into memory, thus we trained the network using a subset of 10k samples only. Validation set and test set are also 10k each. We run the algorithm for 3k iterations with a fixed step size/learning rate of 0.5. The test accuracy is 83.5%. This accuracy is worse than the conventional logistic regression classifier due to the fact that we trained the network on a smaller training set. Since Stochastic GD is designed to be faster than Simple GD, thus we can deploy it using the same Multinomial Logistic Regression but with a larger dataset. With a mini-batch size of 128, we increase the size of training samples to 200k. Validation and test set size are 10k each. We run the algorithm with 3k iterations, and with a constant step size/learning rate of 0.5. The test accuracy is slightly increased to 86.8% due to the fact that a bigger training data set results in better prediction.

We conducted five experiments to evaluate the performance of character classification using various shallow neural networks and well-constructed deep neural networks. In each experiment, 200k samples were used to train the network, and 10k samples were used for validation and testing. We run each algorithm with 30k iterations and with a constant step size/learning rate of 0.5.

In the first experiment, a simple neural network of 1-hidden layer with ReLU and 1024 hidden nodes was trained. This is so called a shallow neural network since it contains only one hidden layer. Fig. 5 shows the detailed architecture, the input image $X_i$ is arranged as a single column vector of $[784 \times 1]$, and passed to the hidden layer. The hidden layer generates an output which is given by equation (10) with the activation function defined in equation (11). The new set of weights $W$ and biases $b$ are used to compute the output scores $Y_i$ as described in equation (2). The output scores are then converted to probabilities using Softmax defined in equation (3). With the current set of weights and biases, we measure the discrepancy (cross-entropy) between the true label for an input pattern and the output produced as defined in equation (4). Thus, the loss function $\mathcal{L}$ is computed by taking the average of cross-entropy over a set of labeled samples as stated equation (5). We then finally optimize $\mathcal{L}$ using Stochastic GD in equation (7). Using this architecture, the test accuracy is improved to 92.7%.

Fig. 6 shows both the mini-batch and validation accuracies of experiment 1 that demonstrates the potential over-fitting problem during the neural network training. The gap between the mini-batch and validation accuracy indicates the amount of overfitting. In practice, when the gap is getting bigger or the validation accuracy starts to decrease, it implies the overfitting, so one can early stop training the network [17]. In Fig. 6, the gap is getting bigger after 20–25k iterations that suggest the validation accuracy may decrease at some point after it. Therefore, we set the number of iteration to be 30k in our experiments.

In the second experiment, we added L2 regularization to the same model in Exp #1. With added L2 regulation, the accuracy is improved to 94.8%. In the third experiment, we added both L2 regularization and a 50% Dropout to the same model in Exp #1 to prevent over fitting. The test accuracy is 94.3% which is similar to Exp #2.

We then proposed to tune the performance of the shallow neural network model to a deep learning model by adding more hidden layers. In Exp #4, we use a multi-layer model
along with the learning rate decay as shown in Fig. 7. Using the learning rate decay is a good practice to lower the learning rate as the training progresses [18, 19]. In particular, this neural network contains 4 hidden layers with ReLU and different numbers of hidden nodes in each layer. Specifically, the number of hidden nodes is 2048, 1024, 1024, and 1024 in each layer respectively. L2 regularization and 50% Dropout were also used in this experiment. The test accuracy is improved to 96.5%.

![Fig. 7. The learning rate decay used in Experiment #4](image)

In the fifth experiment, we proposed a five-layer convolutional neural network model to further improve the classification accuracy. The proposed CNN is shown in Fig. 8 and can be described as following: the first convolutional layer computes the output neurons that are connected to local regions in the input. The ReLU layer applies element wise activation function. The pool layer performs a down sampling operation along the spatial dimensions. The depths of the first, second, third, fourth and fifth convolutional layer are 16, 32, 64, 64 and 256, respectively.

![Fig. 8. The ConvNet architecture used in Experiment 5 and 6](image)

The proposed architecture has 5 convolutional layers with a filter size 4×4, and ReLU layer in between; max pooling layers are inserted after ReLU layer for all layers except for the second convolutional layer. It is then connected to a Fully-Connected Layer of 2-Hidden Layers which have the size of 1024 and 205 nodes. Note that the first layer of the fully-connect part uses ReLU while the second hidden layer uses the ELU activation function. We also use a 50% Dropout, L2 Regularization, and a staircase learning rate decay shown in Fig. 9. The test accuracy for this experiment further improved to 97.5% after 30k iterations.

![Fig. 9. The learning rate decay used in Experiment #5 and #6](image)

In Fig. 10, we show the validation accuracies from various neural networks (Exp #1 to #5) during the training process. The simple neural network of 1-hidden layer in Exp #1 offers the lowest validation accuracy. The validation accuracy improves as we employ L2 regularization and Dropout (Exp #2 and #3). In Exp #4, the deep learning neural network achieves better accuracy than the shallow neural network. Lastly, CNN in Exp #5 offers the best result.

![Fig. 10. The training accuracy of Expemrnt #1 to #5](image)

In addition, we performed another experiment similar to Exp #5 but increase the training size to 400k images. With this change, the test accuracy is further improved to 98%. The validation and test size was also increased to 15ks each and with 95k iterations. Table 1 summary the validation accuracies for the various neural networks from our study.
TABLE I. THE CLASSIFICATION ACCURACY ON THE NOTMNIST DATASET

<table>
<thead>
<tr>
<th>Exp #</th>
<th>Method</th>
<th>Accuracy [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1-hidden layer with ReLU (1024)</td>
<td>92.7</td>
</tr>
<tr>
<td>2</td>
<td>1-hidden layer with ReLU (1024) + L2</td>
<td>94.8</td>
</tr>
<tr>
<td>3</td>
<td>1-hidden layer with ReLU (1024) + L2 + 50% Dropout</td>
<td>94.3</td>
</tr>
<tr>
<td>4</td>
<td>4-hidden layers with ReLU (2048-1024-1024-1024) + L2 + 50% Dropout</td>
<td>96.5</td>
</tr>
<tr>
<td>5</td>
<td>CNN with 200k training set</td>
<td>97.5</td>
</tr>
<tr>
<td>6</td>
<td>CNN with 400k training set</td>
<td>98.0</td>
</tr>
</tbody>
</table>

5 Conclusions

In this paper, we conducted an empirical study for large-scale character recognition using the notMNIST dataset to evaluate the classification accuracy for various neural networks. We showed that deep learning architectures performed better than the conventional neural networks and proposed a five-layer convolutional neural network for this large-scale hard character classification task. The proposed network incorporates rectified linear units (ReLU) and exponential linear unit (ELU) the activation function, and employed L2 regularization and dropout techniques to prevent possible overfitting in training. The result shows that our method can achieve 98% classification accuracy.

All experiments with various architectures were implemented using Python with Tensorflow API. These networks were trained and validated on Amazon Elastic Compute Cloud and with GPU-accelerated utility to reduce the total training time. The proposed architecture and the implementation framework are scalable and cost-effective, and can be easily adapted to other image classification applications using deep neural networks.

6 References


