Character Recognition through Neural Networks

Group #3 Zehui Chen, Shuyang Jiang and Christopher Shaffer

I. Motivation

In some applications, feature spaces are not linearly separable. Consequently, it is impossible to learn the mapping by means of the Perceptron algorithm, which is a useful technique to design a classifier for linearly separable data. In order to overcome this difficulty, some non-linear classifiers were proposed. One powerful method among them is neural networks, which relies on cascading a collection of modified Perceptron units. Neural networks have very different computing approaches from traditional computing machines. The neural networks have an ability to construct the rules of input-output mapping by itself. Thus, the designer of the system does not need to know the internal structure and instructions of the system, or the functional rules like traditional systems.

Neural networks are strong in many scenarios. In our project, we focus on the recognition of characters such as letters and digits through neural networks. Character recognition is a classic pattern recognition problem for which researchers have worked since the early days of computer vision. With today's omnipresence of cameras, the applications of automatic character recognition are broader than ever. It can be used to break a Captcha system, make electronic images of printed documents searchable, etc. Therefore, character recognition is a significant problem that deserves us to study.

II. Model

Neural networks are networks of neurons as in the real biological brain. They are computationally primitive approximations of the real biological neural networks. Every neurons, which are a simple processing unit, is connected in specific ways in order to perform the desired tasks. Next, we will introduce the basic unit in a neural network, neuron.

A. Neurons

A neuron consists of a collection of multipliers, one adder, and a nonlinearity. The input to the first multiplier is fixed at $+1$ and its coefficient is denoted by θ. The coefficients for the remaining multipliers are denoted by $w(m)$ and their respective inputs by $h(m)$. Then we will get an output by the inner product

$$z = h^T w - \theta \tag{1}$$

Then we feed this output into a nonlinearity, called the activation function, to get the final output of this neuron y:

$$y = f(z) = f(h^T w - \theta) \tag{2}$$

There are many choices for this nonlinear function, such as sigmoid, hyperbolic tangent, rectifier and softplus function. Among them, sigmoid function is used quite often. We can see that sigmoid and hyperbolic tangent functions saturate for large $|z|$, which means that a slow down in the speed of learning by neural networks. For rectifier and softplus functions, the learning process will slow down or even stop for negative z.

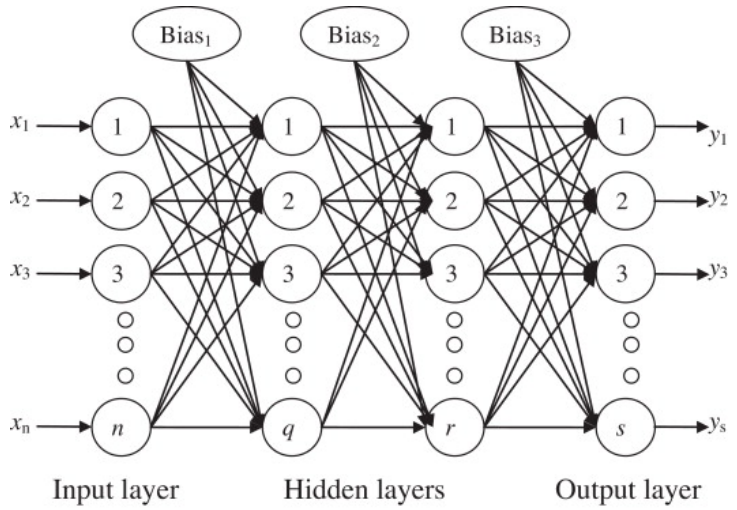


Fig. 1: An example schematic of a feedforward neural networks.

B. Feedforward Neural Networks

There are many architectures for neural networks. One common and powerful implementation is a feedforward multi-layer structure. In this implementation, information flows forward in the network and there is no loop to feed signals from some future layer back to an earlier layer. Fig. 1 illustrates this structure.

We let L denote the number of layers in the network. Usually, networks with more than 3 hidden layers are referred to as deep networks. Between any two layers in the network, there is a collection of combination coefficients W_l that scale the signals arriving from the nodes in the prior layer. We also associate with these layers a bias vector θ_l, containing the coefficients that scale the bias arriving into layer $l+1$ from layer l. We can now examine the flow of signals through the network. we collect the signals prior to and after the activation function at layer l as y_l and z_l, respectively. Then we have a relation of output vectors between two consecutive layers as follows:

$$y_{l+1} = f(z_{l+1}) = f(W_l y_l - \theta_l) \tag{3}$$

Above all, we have a propagation of signals through a feedforward neural network listed as below:

TABLE I: Propagation of signals through a feedforward neural network

start with $y_1 = h$
repeat for $l = 1, ..., L-1$:
$z_{l+1} = W_l y_l - \theta_l$
$y_{l+1} = f(z_{l+1})$
end

C. Recurrent Neural Networks

Besides feedforward neural networks introduced in the handout, there exists another class of neural networks called recurrent neural networks [1]. Recurrent neural networks are built on the same computational units as the feedforward neural networks, but differ in the architecture of how these neurons are connected to one another. Feedforward neural networks are organized in layers, where information flowed unidirectionally from input units to output units. There are no directed cycles in the connectivity patterns. In contrast, RNNs do not have to be organized in layers and directed cycles are allowed. In fact, neurons

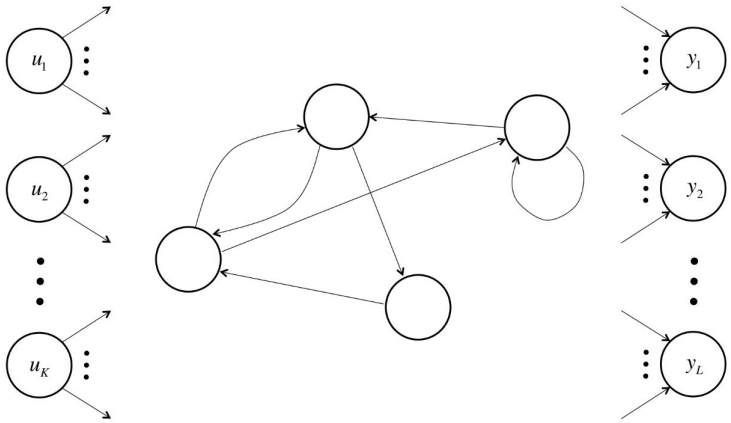


Fig. 2: An example schematic of a recurrent neural networks.

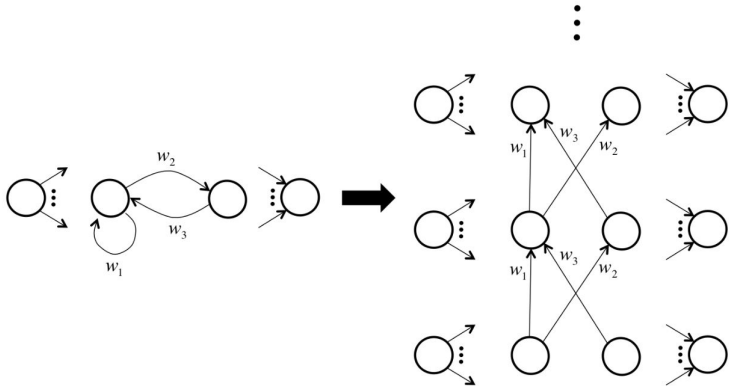


Fig. 3: An example of ”unrolling” recurrent neural networks through time.

are actually allowed to be connected to themselves. An illustration of Recurrent Neural Networks is shown in Fig. 2.

There exists some relation between recurrent neural networks and feedforward neural networks. A clever transformation can be employed to convert our recurrent neural networks into a new structure that's essentially a feedforward neural network. This strategy is called ”unrolling” the recurrent neural networks through time, and an example can be seen in Fig. 3 below:

There are many types of recurrent neural networks [2], incuding long short-term memory (LSTM) network, echo state network (ESN), hopfield network, bidirectional recurrent neural networks, etc. Also, recurrent neural networks have shown great success in many applications [3], such as language modeling and generating text, machine translation, speech recognition, generating image description. In our project, we only focus on the feedforward structure due to its simple structure and widespread use. Thus, we use neural networks to refer to only feedforward neural networks thereafter.

III. Algorithm

A. Backpropagation Algorithm

The backpropagation algorithm is one of the most celebrated procedures for training neural networks. As a byproduct, the algorithm will help provide an effective recursive construction for evaluating how the performance of the network varies in response to small changes in its internal coefficients.

In order to get the backpropagation algorithm, we need to consider the following regularized least-squares problems:

$$\{W_l^o, \theta_l^o\} \triangleq \arg\min_{W_l, \theta_l} \sum_{l=1}^{L-1} \rho \|W_l\|_F^2 + \frac{1}{N} \sum_{n=0}^{N-1} \|\gamma_n - y_n\|^2 \tag{4}$$

as we already know, regularization helps avoid over-fitting and improves the generalization ability of the network. Other forms of regularization are possible, including l_1-regularization, as well as other risk functions like cross-entropy function. Their main purpose is to avoid a problem that causes a slow down in the learning rate of some nodes in the network.

When solving (4), we need to evaluate two gradients relative to the individual entries of $\{W_l, \theta_l\}$ as follows

$$\frac{\partial J_{\text{emp}}(W, \theta)}{\partial w_{ij}^{(l)}} \quad \text{and} \quad \frac{\partial J_{\text{emp}}(W, \theta)}{\partial \theta_l(j)} \tag{5}$$

where $J_{\text{emp}}(W, \theta) \triangleq \sum_{l=1}^{L-1} \rho \|W_l\|_F^2 + \frac{1}{N} \sum_{n=0}^{N-1} \|\gamma_n - y_n\|^2$. To pursue a systematic and structured derivation of the above gradients, we introduce sensitivity factors:

$$\delta_l(j) \triangleq \frac{\partial \|\gamma - y\|^2}{\partial z_l(j)} \tag{6}$$

Here, we drop the time subscript n to simplify the notation.

First, we have to derive a recursive update for the vector δ_l. we note that

$$\begin{aligned}
\delta_l(j) \triangleq & \frac{\partial \|\gamma - y\|^2}{\partial z_l(j)} \\
= & \sum_{k=1}^{n_{l+1}} \frac{\partial \|\gamma - y\|^2}{\partial z_{l+1}(k)} \frac{\partial z_{l+1}(k)}{\partial z_l(j)} \\
= & \sum_{k=1}^{n_{l+1}} \delta_{l+1}(k) \frac{\partial z_{l+1}(k)}{\partial z_l(j)} \\
= & \left(\sum_{k=1}^{n_{l+1}} \delta_{l+1}(k) w_{jk}^{(l)} \right) f'(z_l(j)) \\
= & f'(z_l(j)) \left(w_j^{(l)}\right)^T \delta_{l+1}
\end{aligned} \tag{7}$$

We use the vector notation and get that

$$\boxed{\delta_l = f'(z_l) \odot (W_l^T \delta_{l+1}) \quad l = 2, 3, ..., L-1} \tag{8}$$

For δ_L, we have that

$$\begin{aligned}
\delta_L(j) = & \sum_{k=1}^{Q} \frac{\partial \|\gamma - y\|^2}{\partial y(k)} \frac{\partial y(k)}{\partial z(j)} \\
= & \sum_{k=1}^{Q} 2(y(k) - \gamma(k)) \frac{\partial y(k)}{\partial z(j)} \\
= & 2(y(j) - \gamma(j)) f'(z(j))
\end{aligned} \tag{9}$$

Similarly, we also use the vector notation to write as

$$\boxed{\delta_L = 2(y - \gamma) \odot f'(z)} \tag{10}$$

Next, we are in a postion to partial derivatives of $\|\gamma-y\|^2$ relative to the combination weights. Following similar arguments to the above, we have that

$$\begin{aligned}\frac{\partial\|\gamma-y\|^2}{\partial w_{ij}^{(l)}}=&\sum_{k=1}^{n_{l+1}}\frac{\partial\|\gamma-y\|^2}{\partial z_{l+1}(k)}\frac{\partial z_{l+1}(k)}{\partial w_{ij}^{(l)}}\\=&\delta_{l+1}(j)y_l(i)\end{aligned}\tag{11}$$

from above, we can get the expression of one gradient in (5) as follows

$$\boxed{\frac{\partial J_{\text{emp}}(W,\theta)}{\partial W_l}=2\rho W_l+\frac{1}{N}\sum_{n=0}^{N-1}\delta_{l+1,n}y_{l,n}^T}\tag{12}$$

At last, we can use the same method to get partial derivatives of $\|\gamma-y\|^2$ relative to the bias weights and get that

$$\begin{aligned}\frac{\partial\|\gamma-y\|^2}{\partial\theta_l(i)}=&\sum_{k=1}^{n_{l+1}}\frac{\partial\|\gamma-y\|^2}{\partial z_{l+1}(k)}\frac{\partial z_{l+1}(k)}{\partial\theta_l(i)}\\=&-\delta_{l+1}(i)\end{aligned}\tag{13}$$

and

$$\boxed{\frac{\partial J_{\text{emp}}(W,\theta)}{\partial\theta_l}=-\frac{1}{N}\sum_{n=0}^{N-1}\delta_{l+1,n}}\tag{14}$$

In summary, we can get the backpropagation algorithm as listed below

TABLE II: The procedure of backpropagation algorithm

start with traing data $\{\gamma_n,h_n\},n=0,1,...,N-1.$
$D_l=2\rho W_l,d_l=0_{n_{l+1}},l=1,2,...,L-1.$
repeat for $n=0,1,...,N-1:$
  (forward pass)
    feed h_n into the network and compute $\{z_{l,n},y_{l,n}\}$ using Table I.
    compute terminal sensitivity vector: $\delta_{L,n}=2(y_{L,n}-\gamma_n)\odot f'(z_{L,n}).$
  (backward pass)
  for $l=L-1,...,3,2,1$
    $\delta_{l,n}=f'(z_{l,n})\odot(W_l^T\delta_{l+1,n})$
    $D_l=D_l+\frac{1}{N}\delta_{l+1,n}y_{l,n}^T$
    $d_l=d_l-\frac{1}{N}\delta_{l+1,n}$
  end
end
$\frac{\partial J_{\text{emp}}(W,\theta)}{\partial W_l}=D_l$
$\frac{\partial J_{\text{emp}}(W,\theta)}{\partial\theta_l}=d_l$

B. Stochastic-Gradient Backpropagation Algorithm

Now, we can employ the backpropagation algorithm to train a neural network through a stochastic-gradient implementation.

For initialization, it is customary to select the bias coefficients $\{\theta_{l,-1}(i)\}$ randomly by following $\mathcal{N}(0,1)$. The combination weights $\{w_{ij,-1}^{(l)}\}$ are also selected randomly according to a Gaussian distribution $\mathcal{N}(0,\frac{1}{\sqrt{n_l}})$ or a uniform distribution $[-\frac{1}{\sqrt{n_l}},\frac{1}{\sqrt{n_l}}]$, $[-\frac{\sqrt{6}}{\sqrt{n_l+n_{l+1}}},\frac{\sqrt{6}}{\sqrt{n_l+n_{l+1}}}]$, $[-\frac{4\sqrt{6}}{\sqrt{n_l+n_{l+1}}},\frac{4\sqrt{6}}{\sqrt{n_l+n_{l+1}}}]$. These selections are

meant to ensure that during the initial stages of training, information can flow reliably forward and backward in the network away from saturation and the difficulties caused by the vanishing gradient problem.

Here we list the procedure of stochastic-gradient backpropagation algorithm as follows

TABLE III: The procedure of stochastic-gradient backpropagation algorithm

For each training point$\{\gamma_n, h_n\}, n = 0, 1, ..., N-1$.
  feed h_n into the network $\{W_{l,n-1}, \theta_{l,n-1}\}$ and compute $\{z_{l,n}, y_{l,n}\}$ using Table I.
  compute $\delta_{L,n} = 2(y_{L,n} - \gamma_n) \odot f'(z_{L,n})$.
  repeat for $l = L-1, ..., 3, 2, 1$
    $\delta_{l,n} = f'(z_{l,n}) \odot (W_{l,n-1}^T \delta_{l+1,n})$
    $W_{l,n} = (1 - 2\mu\rho)W_{l,n-1} - \mu\delta_{l+1,n} y_{l,n}^T$
    $\theta_{l,n} = \theta_{l,n-1} + \mu\delta_{l+1,n}$
  end
end

We can also train the feedforward neural networks by employing a mini-batch implementation. The batch samples can be chosen in various ways, e.g., as the most recent B samples in a streaming implementation or randomly from within the entire N samples.

C. Motivation for Other Variant Algorithms

Now we observe the update of the neural networks $\{W_{l,n}, \theta_{l,n}\}$, it can be seen that these updates will be affected by the entries of $\delta_{l,n}$. We further find that these entries of $\delta_{l,n}$ will involve a product of derivatives of the activation function, $f'(\cdot)$, at successive output signals, namely,

$$\text{entries of } \delta_{l,n} \propto f'(z_{l,n}) f'(z_{l+1,n}) ... f'(z_{L,n}) \tag{15}$$

It is seen that products of a large collection of derivative values for these activation functions can result in a small number, especially when the nodes in the output layer are close to saturation in which case $f'(z_{L,n})$ will already be close to zero. This problem is known as the vanishing gradient problem.

As the entries of $\delta_{l,n}$ become relatively small and, consequently, the updates to $W_{l,n-1}$ and $\theta_{l,n-1}$ will be slow. Observe that this effect is magnified for the earlier layers in the network due to the backward nature of the recursion for $\delta_{l,n}$: as we move further back in the layers, more terms are included in the product and it will become more likely for the product to assume small values. This means that earlier layers in the network will end up learning at a relatively slower rate compared to the later layers.

For similar reasons, the learning process in the network is also slowed down when some nodes are saturated. This is because the derivatives of the corresponding activation functions will be close to zero, and the product will again be close to zero. As already indicated, saturation is more likely to occur at the output nodes, thus leading to small values for $f'(z_{L,n})$. In order to eliminate the effect of the derivatives of the activation functions at the output nodes, we will consider two strategies: 1) adjustments to the network structure, or 2) the modification of the risk function.

D. Variant Algorithm by Adjusting Network Structure

One popular adjustment of network structure is in the form of a softmax implementation. It only changes the structure of the output layer in the network. The activation functions in this layer are replaced by the following normalization step:

$$y_n(q) \triangleq e^{z_n(q)} \left(\sum_{k=1}^{Q} e^{z_n(k)} \right)^{-1} \tag{16}$$

Consequently, we will find that the listings of the stochastic-gradient backpropagation algorithms remains intact except for the expression for the terminal sensitivity vector, $\delta_{L,n}$, which is replaced by

$$\delta_{L,n} = x - (\mathbf{1}^T x) y_{L,n} \tag{17}$$

where $x \triangleq 2(y_{L,n} - \gamma_n) \odot y_{L,n}$. It can be seen that $f'(z_{L,n})$ is eliminated from the expression.

E. Variant Algorithm by modifying risk function

There are several variant algorithms that are derived by modifying the risk function. One of them is in the cross-entropy formulation. Its empirical risk[1] is

$$J_{\text{emp}}(W,\theta) \triangleq \sum_{l=1}^{L-1} \rho \|W_l\|_F^2 - \frac{1}{N} \sum_{n=0}^{N-1} \sum_{q=1}^{Q} \ln\Big(y_n(q)^{\gamma_n(q)} (1 - y_n(q))^{(1-\gamma_n(q))} \Big) \tag{18}$$

Then we will get that the only change to the stochastic-gradient backpropagation algorithms is that $\delta_{L,n}$ is derived by

$$\delta_{L,n} = y_{L,n} - \gamma_n \tag{19}$$

Another formulation uses the logistic risk as follows:

$$J_{\text{emp}}(W,\theta) \triangleq \sum_{l=1}^{L-1} \rho \|W_l\|_F^2 + \frac{1}{N} \sum_{n=0}^{N-1} \ln\Big(1 + e^{-\gamma_n y_n} \Big) \tag{20}$$

IV. Simulation

A. Datasets

In this project, we trained a neural network to recognize images of English capital letters (A-Z) and numbers (0-9). We used the Chars74k dataset [4], which includes images of printed, handwritten, and computer generated characters of different fonts. The "natural scenes" subset of the Chars74K dataset, includes photos of text on signs, advertisements, and product labels with different fonts, camera position, illumination, and image resolution. The "handwritten characters" and "handwritten digits" subsets include hand drawn letters and numbers using a tablet PC. The "computer generated fonts" subset includes synthesized characters from computer fonts. The number of training and testing samples for each data subset is listed in the table below.

Dataset	N_{train}	N_{test}
Natural scenes	4702	500
Handwritten characters	1170	260
Handwritten digits	450	100
Computer generated fonts	23816	2600

The images were converted into feature vectors using the following process. First, the images were converted into 32x32 pixel bitmap images by cropping or adding white space, centering, and downsampling. Then, using the process outlined by the Optical Recognition of Handwritten Digits Data Set [5], the 32x32 pixel bitmaps were divided into blocks of 4x4 and the number of "on" pixels were counted in each block. This defines an 8x8 matrix, in which each element is an integer in the range 0-16, referring to the number of "on" pixels in each block. This was done to reduce dimensionality, and reduce sensitivity to small variations. Last, the 8x8 matrix was vectorized into a 64 element feature vector, $h_n \in \mathbb{R}^M$, where $M = 64$.

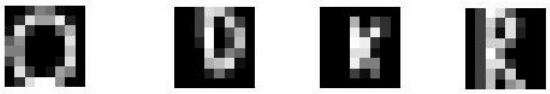
The classification of the letters was defined as a multiclass classification problem, and the class variable was defined as a vector, $\gamma_n \in \mathbb{R}^Q$, where $Q = 26$, for natural scenes, handwritten characters, and computer generated fonts datasets, is the number of letters in the English alphabet, and $Q = 10$ is the number of unique numbers (0-9). Each entry of this vector variable corresponds to one class. When h_n corresponds to some class c, the c–th entry of γ_n will be one, while all other entries will be zero.

[1]There may be a typo in (47.1051) in the handout. A factor of $1/N$ is missing is the second item of $J_{\text{emp}}(W,\theta)$.

(a) Natural scenes

(b) Handwritten characters

(c) Computer generated fonts



(d) Feature vector values after image processing

Fig. 4: Examples of images of letters from datasets, (a), (b), and (c), and feature vectors values after image processing, (d)

B. Training Algorithm

We tested a variety of learning algorithms for this project, which are described in detail in Section III. We initially tested the sigmoid activation function, but found that saturation of large positive z values resulted in small values for $f'(z)$, resulting in slow down in learning from the vanishing gradient problem. As a result, we also tested the softplus activation function defined as:

$$f(z) = \ln(1 + e^z) \tag{21}$$

Other variations of the algorithm we tested include softmax training and cross-entropy training.

For initialization, the bias coefficients $\{\theta_{l,-1}(i)\}$ were randomly generated according to the Gaussian distribution $\mathcal{N}(0, 1)$. When a sigmoid activation function was used, which includes the cross-entropy training case, the combination weights $\{w^{(l)}_{ij,-1}\}$ were generated randomly according to a uniform distribution $\left[-\frac{4\sqrt{6}}{\sqrt{n_{l+1}}+\sqrt{n_l}}, \frac{4\sqrt{6}}{\sqrt{n_{l+1}}+\sqrt{n_l}}\right]$. When a softplus activation function or softmax training were used, the weights were generated with a uniform distribution $\left[-\frac{1}{\sqrt{n_l}}, \frac{1}{\sqrt{n_l}}\right]$. We used the feedforward algorithm as described in Table I, and the stochastic-gradient backpropagation algorithm, as described in Table III. When softmax was implemented, we followed the modifications to the stochastic-gradient backpropagation algorithm described in Section III-D. Similarly, the cross-entropy training follows the modifications described in Section III-E.

For each training pass, we randomized the order of training samples. Then, we trained on a single sample, at each iteration, n, and repeated for N_{train} iterations, where N_{train} was the total number of training samples. The performance was then tested using the procedure outlined in Section IV-C. Testing Algorithm, below. For each subsequent training pass, the trained weight and bias values $\{W_{N_{\text{train}}-1}, \theta_{N_{\text{train}}-1}\}$, were used as the initial values $\{W_{-1}, \theta_{-1}\}$, and the sample order was randomized again. This process was repeated for some number of passes, P.

C. Testing Algorithm

After the network was trained on all samples in the testing dataset, the performance of the network was tested a separate testing dataset. For each sample in the testing dataset, the output of the final layer y_L was calculated using the feedforward algorithm with trained weights and biases W_l and θ_l. Then, the output y_L was converted into an estimated class vector γ' using the following

$$\begin{cases} \gamma'(q) = 1, & \text{if } y_L(q) = \max(\{y_L(1), ..., y_L(n_L)\}) \\ \gamma'(q) = 0, & \text{if } y_L(q) \neq \max(\{y_L(1), ..., y_L(n_L)\}) \end{cases} \tag{22}$$

This class vector γ' was compared with the actual class vector, γ to determine the error rate over all samples in the testing dataset:

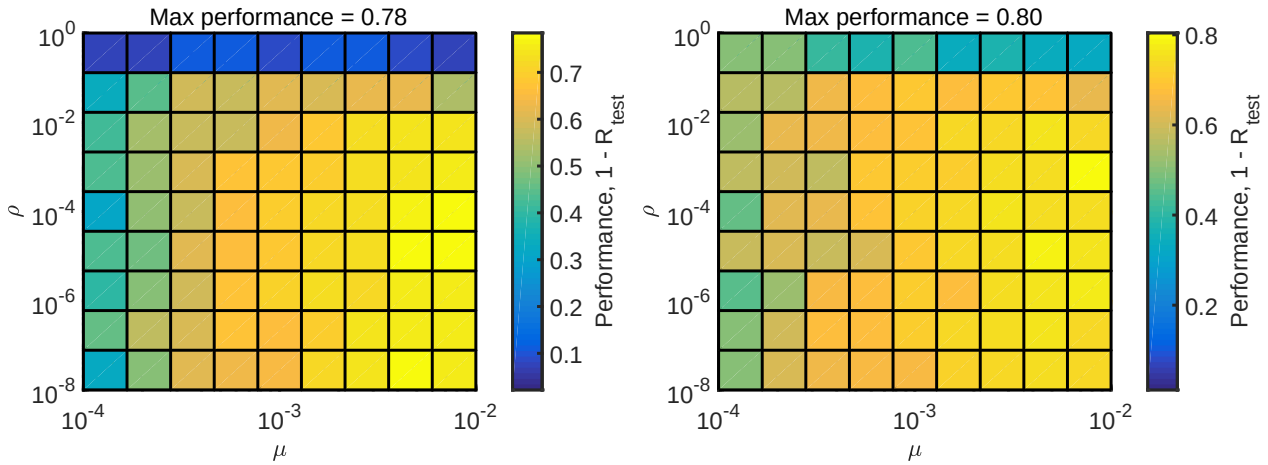
$$R_{\text{test}}(W, \theta) = \frac{1}{N_{\text{test}}} \sum_{n=0}^{N_{\text{test}}-1} \mathbb{I}\Big[\gamma(n) \neq \gamma'(n)\Big] \tag{23}$$

This process was repeated for a number of training passes, with the weights and biases retained after each pass, and the empirical error rate was recorded as a function of number of passes, P.

V. Results

A. Parameter Optimization

1) Number of Hidden Layers: To determine the number of hidden layers to use, we tested the performance of the algorithm using 1 and 2 hidden layers. We performed an optimization over the regularization factor ρ, and the step-size μ, to determine the maximum performance and optimum values of each parameter. We used the "natural scenes" dataset for uppercase characters. These parameters were varied over a grid of logarithmically spaced values. For the 2 hidden layer case, $n_2 = n_3 = 100$, and also for the 1 hidden layer case, $n_2 = 100$. Both tests used the softmax algorithm, which we determined has the best performance. The number of passes over all samples was $P = 10$, to reduce runtime over the large number of points tested. Fig. 5 shows the result of the test. The 2 hidden layer case had a maximum performance of $1 - R_{\text{test}} = 0.78$,



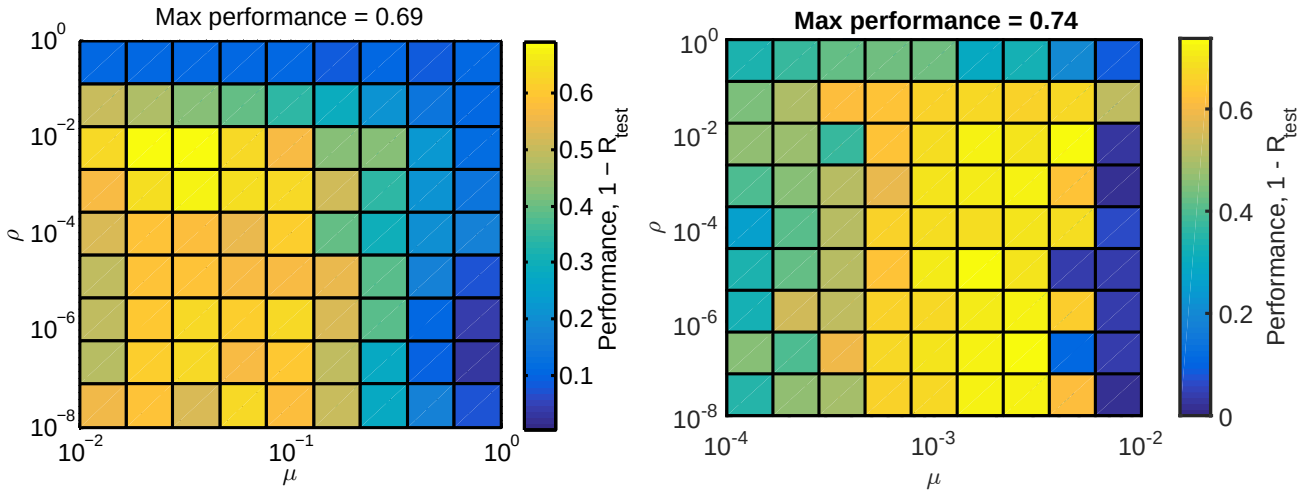
(a) 2 Hidden Layers

(b) 1 Hidden Layer

Fig. 5: Heatmaps of μ versus ρ versus $1 - R_{test}$ on "natural scenes" dataset for uppercase characters, using softmax algorithm, for $P = 10$

and the 1 hidden layer case had a maximum performance of 0.80. The slightly worse performance of the 2 hidden layer case may be due to the vanishing gradient problem, from saturation of output values at successive layers. We conclude that 1 hidden layer results in better performance for this experiment, given the parameters tested. Additionally, the 1 hidden layer case is computationally simpler, resulting in faster runtime.

2) Activation Function: Similar to the test to determine the number of layers, we performed an optimization over the regularization factor ρ, and the step-size μ, to determine the maximum performance and optimum values of each parameter. We used the "natural scenes" dataset for uppercase characters. These parameters were varied over a grid of logarithmically spaced values. The range of μ shown for the sigmoid function is different than for the softplus function, because we found emipirically that the optimums for each are within the ranges shown. The number of hidden nodes n_2 was 100. The number of passes over all samples was $P = 10$, to reduce runtime over the large number of points tested. Fig. 6 shows the result of the test. In the range of parameters tested, the sigmoid activation function had a maximum performance of $1 - R_{\text{test}} = 0.69$, and the softplus activation function had a maximum performance of 0.74. This is likely due to the vanishing gradient problem for positive output values when using the sigmoid function. We conclude that the softplus activation function results in better performance for this dataset, given the parameters tested.



(a) Sigmoid activation function

(b) Softplus activation function

Fig. 6: Heatmaps of μ versus ρ versus $1 - R_{test}$ on "natural scenes" dataset for uppercase characters, for $P = 10$

3) Hidden Layer Nodes: The number of nodes in the input layer is defined as the number of image pixels, $n_1 = M = 64$, and the number of nodes in the output layer is defined as the number of classes, $n_3 = Q = 26$ for the "natural scenes" dataset. The number of nodes in the hidden layer, n_2 is not as trivially defined, and was optimized empirically. The performance of the algorithm was tested using different values for n_2. We used the testing protocol described in Section IV-B. Training Algorithm and Section Section IV-C. Testing Algorithm. The number of passes over all samples was $P = 10$. The regularization factor was $\rho = 1 \times 10^{-4}$, and the step-size was $\mu = 1 \times 10^{-3}$, because these were found to give the optimum performance from the optimization test. The values for n_2 were varied over a range $[20, 200]$. Fig. 7 shows the result of the n_2 optimization test.

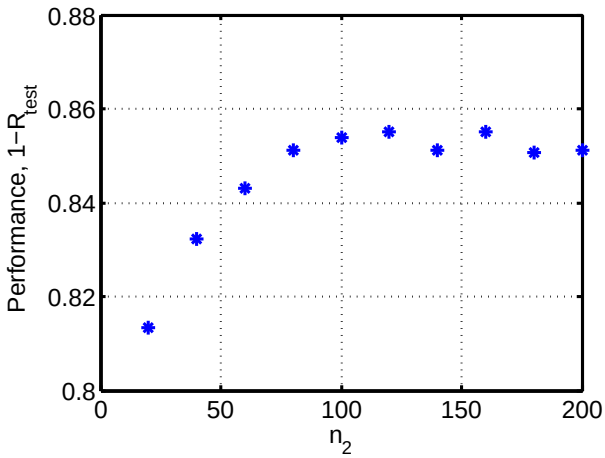
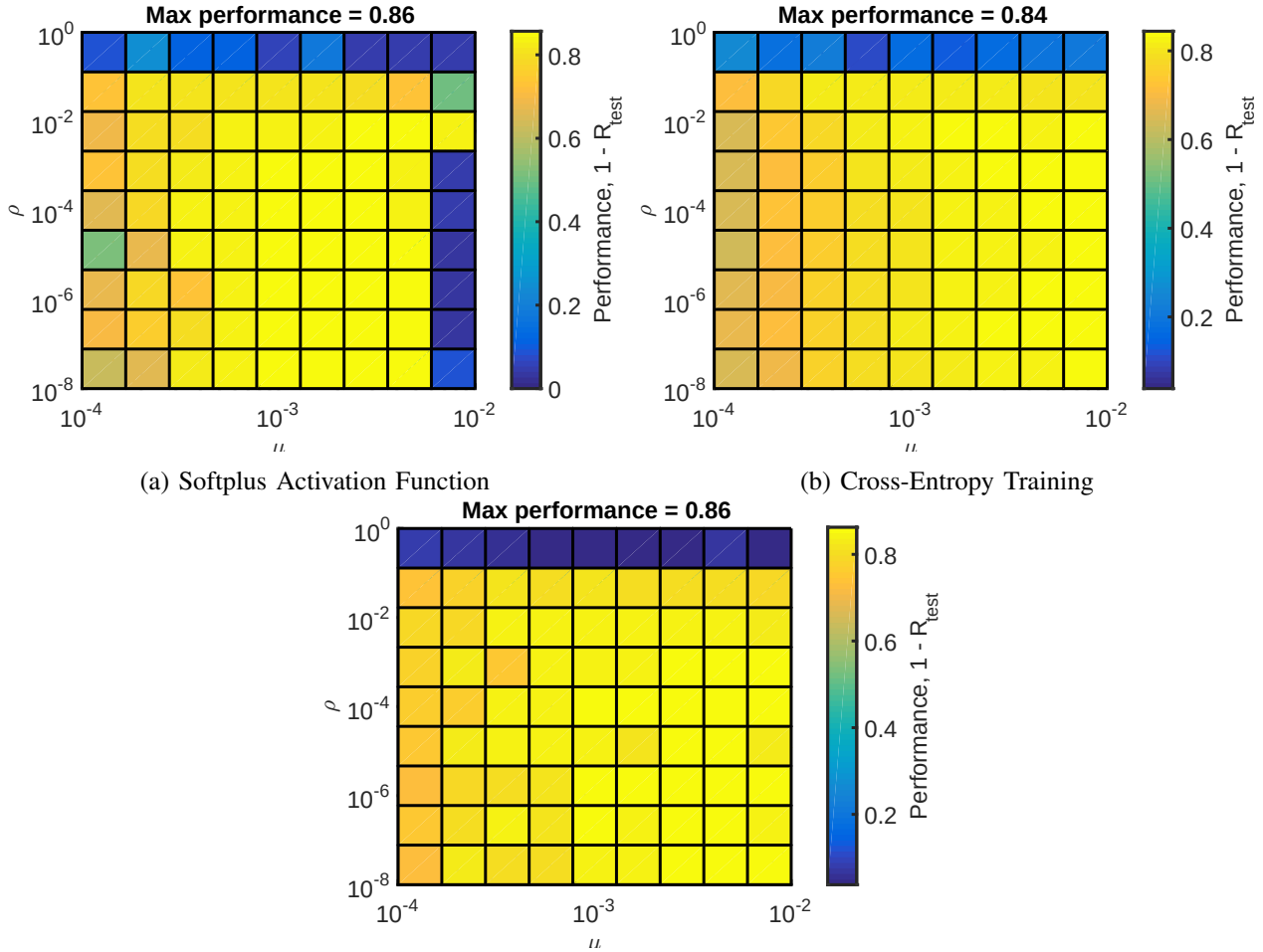


Fig. 7: The number of hidden nodes versus performance, n_2 , for $P = 10$.

At $n_2 = 100$, the performance improvement saturates, with no significant improvement with increasing values of n_2 We choose to use $n_2 = 100$ for all other tests because it gives approximately the best performance, while having a shorter runtime than larger values of n_2. Overfitting may also be a problem at larger values of n_2.

B. Performance comparison

In this subsection, we compare the classification performance for three algorithm variants: softplus activation function, softmax, and cross-entropy. We use the computer generated fonts dataset as the testing dataset first, because it has the highest performance due to its large sample size. From Fig. 8, we see that the softplus and softmax algorithms have a maximum performance, $1 - R_{\text{test}} = 0.86$, while the cross-entropy algorithm has $1 - R_{\text{test}} = 0.84$. This shows that the softplus and softmax algorithms are superior to the



(a) Softplus Activation Function

(b) Cross-Entropy Training

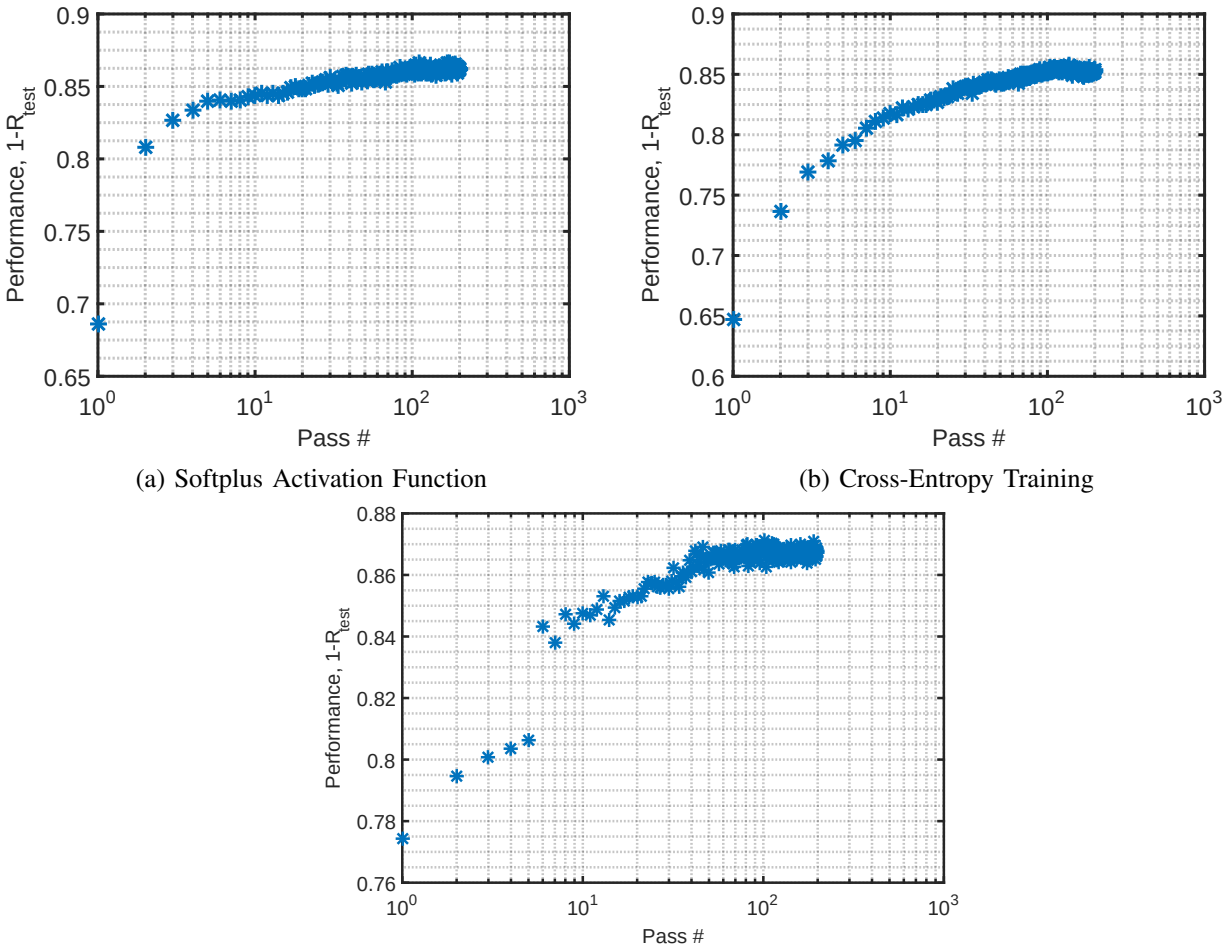
(c) Softmax Training

Fig. 8: Parameter heatmap for computer generated fonts, for P=10

cross-entropy algorithm. For the sake of runtime, these experiments only used $P = 10$, so the final optimum values after many passes may be different. Fig. 9, shows the performance as a function of pass number, P, for P_{max} = 200. After the performance improvement saturates, we find the maximum performance of softmax to be $1 - R_{\text{test}} = 0.87$, of softplus to be $1 - R_{\text{test}} = 0.86$, of cross-entropy to be $1 - R_{\text{test}} = 0.85$. The results are close, but we can see that softmax training is superior than the other two algorithms.

We also tested the performance of these algorithms in the other three datasets, natural scenes, handwritten characters and handwritten digits. Due to limited space, we put the parameter heatmaps Fig. 11, 12 and 13 in the Appendix. For all datasets, we find that softmax training has performance equal to or better than the softplus activation function, and strictly better performance than cross-entropy training. Overall, the softmax algorithm had the best performance in every test we ran.

At last, we show the performance of softmax training for each character type from each dataset in Fig. 10. It is difficult to evaluate the intrinsic difficulty of recognizing each dataset, since the number of training and testing samples in each is variable. The performance correlates with sample size, and the best and



(a) Softplus Activation Function

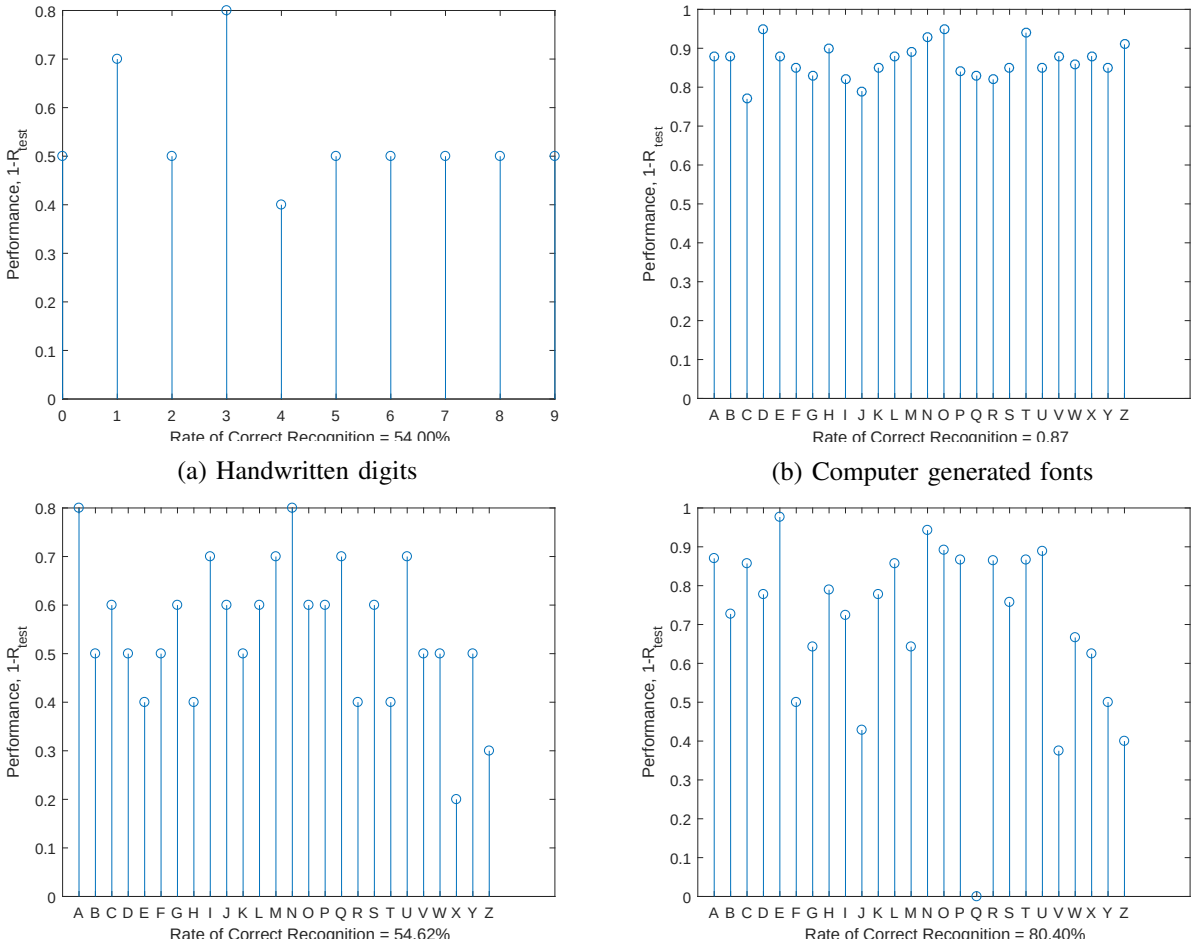
(b) Cross-Entropy Training

(c) Softmax Training

Fig. 9: P versus $1 - R_{\text{test}}$ computer generated fonts, for $P_{\text{max}} = 200$

worst performing datasets also had the largest and smaller numbers of samples, respectively. Similarly, the letter, "Q" from the natural scenes dataset stands out as having the worst performance, with $1 - R_{\text{test}} = 0$, and this had a disproportionately small number of samples compared to the other letters. This comparison with different datasets highlights the need for a sufficiently large training sample size to achieve high performance.

Lastly, although the performance varies due to the aforementioned variation of sample size, this comparison of datasets does show the robustness of the algorithm to a large variation of data types. Despite sample variation from angle or blur in the natural scenes dataset, font variation in the computer generated fonts dataset, or human variation of handwriting in the handwritten characters dataset, the algorithm is able to robustly identify the characters correctly, with high success rate.



(a) Handwritten digits

(b) Computer generated fonts

(c) Handwritten characters

(d) Natural scenes, uppercase characters

Fig. 10: Stemplots of performance, $1 - R_{\text{test}}$, for $P = 200$, with softmax training

VI. Conclusion

In this project, we focused on the application of feedforward neural networks with backpropagation to character recognition. The model of feedforward neural networks was studied in detail along with the backpropagation algorithm, which is important in evaluating how the network changes in response to small changes. We also studied the algorithms for training which include stochastic-gradient backpropagation algorithm and its two variants. The softmax training changes the activation functions in the output layer into a normalization step. The cross entropy training modifies the risk function with a cross-entropy formulation. The two variants were considered in order to counter the slowdown of learning thus improve the performance. The three algorithms were implemented and tested with the Chars74k dataset [4]. Four groups of data from the dataset were used for testing, including ”natural scenes” uppercase characters, handwritten characters, handwritten digits and computer generated fonts. The performance of certain algorithms on certain data was measured in terms of $1 - R_{test}$ which is the rate of correct recognition by the trained network, and displayed on a heatmap spanned by its two parameters, the regularization factor ρ and the step size μ. We also compared the performance of different activation functions, softplus and sigmoid, for the stochastic-gradient training algorithm. As a result, we found out softplus outperforms sigmoid because it prevents saturation. Among the three algorithms, the softmax algorithm has better performance than the other two. Among the four datasets, the computer generated font dataset has the best performance because the neural network can be trained sufficiently on its large number of sample. When the neural network is

trained on computer generated uppercase font data with the softmax algorithm, the performance can reach 0.87. This result means the network can correctly recognize 87% of the testing characters.

VII. Contribution

All three members contribute equally for this project.

Zehui Chen:

- data preprocessing
- stochastic-gradient backpropagation algorithm
- $real\_testing.m$ files for result display
- testing, editing and evaluation

Shuyang Jiang:

- softmax training algorithm with its associated feedforward and backpropagation functions
- cross-entropy training algorithm with its associated feedforward and backpropagation functions
- testing, editing and evaluation

Christopher Shaffer:

- feedforward function for stochastic-gradient backpropagation
- $training\_test\_param.m$ files for parameter optimization
- stochastic-gradient backpropagation with softplus activation
- testing, editing and evaluation

References

[1] Nikhil Buduma (January 11, 2015), *"A Deep Dive into Recurrent Neural Nets"*.
[2] Recurrent neural network, *wikipedia*.
[3] Denny Britz (September 17, 2015), *"Recurrent Neural Networks Tutorial, Part 1 – Introduction To RNNs"*, wildml.
[4] T. E. de Campos, B. R. Babu and M. Varma. Character recognition in natural images. In Proceedings of the International Conference on Computer Vision Theory and Applications (VISAPP), Lisbon, Portugal, February 2009.
[5] Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

APPENDIX

A. Input/Output Definition for Important MATLAB Functions

1) Initialization: The following functions initialize the starting combination weights and bias vectors for the neural network.
$[W, \theta] = network\_initial(L, N_l)$
$[W, \theta] = network\_initial\_sp(L, N_l)$

The first function is used to initialize combination weights and bias vectors for network using a sigmoid function. W is chosen uniformly from $\left[-\frac{\sqrt{6}}{\sqrt{n_{l+1}}+\sqrt{n_l}}, \frac{\sqrt{6}}{\sqrt{n_{l+1}}+\sqrt{n_l}}\right]$, θ is chosen from a Gaussian distribution $N(0,1)$. The second function is used to initialize combination weights and bias vectors for network using a softplus activation function. W is chosen uniformly from $\left[-\frac{1}{\sqrt{n_l}}, \frac{1}{\sqrt{n_l}}\right]$, θ is chosen from a Gaussian distribution $N(0,1)$.

Input:
L: a scalar value denoting the number of layers (including input layer, output layer and hidden layers)
N_l: a vector with size $1 \times L$ containing the number of nodes in the l_{th} layer in $N_l(l)$, namely n_l. Note that n_1 is the length of the feature vector and n_L is the length of output vector

Output:
W:a 3d array with size $max(n_l) \times max(n_l) \times (L-1)$ containing a collection of matrix W_l, $W(1:n_{l+1}, 1:n_l, l)$ denoting the combination weights from layer l to layer l+1 with size $n_{l+1} \times n_l$. Each combination weight matrix is generated randomly according to equation 47.1025 on page 2488 of the reference under the section of initialization. All unused array elements are padded with 0.
θ: a matrix with size $max(n_l) \times (L-1)$ contain a collection of column vector θ_l, $\theta(1:n_{l+1}, l)$denoting the bias vector from layer l to layer l+1 with size $n_{l+1} \times 1$.Each bias vector is generated according to equation 47.1025 on page 2488 of the reference under the section of initialization. All unused matrix elements are padded with 0.

2) Feed-forward Path: The following functions use the current combination weights and bias vector to compute the output for each layer(before and after activation).
$[Z, Y] = feedforward(W, \theta, h, L, N_l)$
This function uses sigmoid function as activation function.
$[Z, Y] = feedforward\_sp(W, \theta, h, L, N_l)$
This function uses softplus as activation function.
$[Z, Y] = feedforward\_sm(W, \theta, h, L, N_l)$
This function is adpated to use the softmax algorithm.

Input:
W:a 3d array with size $max(n_l) \times max(n_l) \times (L-1)$ containing a collection of matrix W_l, $W(1 : n_{l+1}, 1 : n_l, l)$ denoting the combination weights from layer l to layer l+1 with size $n_{l+1} \times n_l$. All unused array elements are padded with 0.
θ: a matrix with size $max(n_l) \times (L-1)$ contain a collection of column vector θ_l, $\theta(1 : n_{l+1}, l)$denoting the bias vector from layer l to layer l+1 with size $n_{l+1} \times 1$.. All unused matrix elements are padded with 0.
h:a vector containing the current used feature vector with size $n_1 \times 1$.
L: a scalar value denoting the number of layers (including input layer, output layer and hidden layers)
N_l: a vector with size $1 \times L$ containing the number of nodes in the l_{th} layer in $N_l(l)$, namely n_l. Note that n_1 is the length of the feature vector and n_L is the length of output vector

Output:
Z: a matrix with size $max(n_l) \times L$ contain a collection of column vector z_l, $Z(1 : n_l, l)$denoting the before activation output of layer l with size $n_l \times 1$. Note that z_1 does not have meaning and is left to be zeros for indexing consistency. Other unused elements are set to 0.
Y: a matrix with size $max(n_l) \times L$ contain a collection of column vector y_l, $Y(1 : n_l, l)$denoting the before activation output of layer l with size $n_l \times 1$. Note that y_1 is set equal to the feature vector h and y_2 is the output of the neural network.

3) Stochastic-gradient backpropagation algorithm: The following function use the current state of neural network to update the combination weights and bias vector.
$[W_{new}, \theta_{new}] = grad\_back(Z, Y, L, N_l, \rho, \mu, \gamma, W_{old}, \theta_{old})$
This function performs the stochastic-gradient backpropagation algorithm with sigmoid as activation function.
$[W_{new}, \theta_{new}] = grad\_back\_sp(Z, Y, L, N_l, \rho, \mu, \gamma, W_{old}, \theta_{old})$
This function performs the stochastic-gradient backpropagation algorithm with softplus as activation function.
$[W_{new}, \theta_{new}] = grad\_back\_sm(Z, Y, L, N_l, \rho, \mu, \gamma, W_{old}, \theta_{old})$
This function adapts the Softmax stochastic-gradient backpropagation algorithm.
$[W_{new}, \theta_{new}] = grad\_back\_ce(Z, Y, L, N_l, \rho, \mu, \gamma, W_{old}, \theta_{old})$
This function adapts the Cross-entropy mini-batch backpropagation algorithm.

Input:
Z: a matrix with size $max(n_l) \times L$ contain a collection of column vector z_l, $Z(1 : n_l, l)$denoting the before activation output of layer l with size $n_l \times 1$. Note that z_1 does not have meaning and is left to be zeros for indexing consistency. Other unused elements are set to 0.
Y: a matrix with size $max(n_l) \times L$ contain a collection of column vector y_l, $Y(1 : n_l, l)$denoting the before activation output of layer l with size $n_l \times 1$. Note that y_1 is set equal to the feature vector h and y_2 is the output of the neural network.
L: a scalar value denoting the number of layers (including input layer, output layer and hidden layers)
N_l: a vector with size $1 \times L$ containing the number of nodes in the l_{th} layer in $N_l(l)$, namely n_l. Note that n_1 is the length of the feature vector and n_L is the length of output vector
ρ:the regularization factor chosen by the user.
μ: the step size chosen by the user.
W_{old}:the un-updated 3d array with size $max(n_l) \times max(n_l) \times (L-1)$ containing a collection of matrix W_l, $W(1 : n_{l+1}, 1 : n_l, l)$ denoting the combination weights from layer l to layer l+1 with size $n_{l+1} \times n_l$. All unused array elements are padded with 0.
θ_{old}: a un-updated matrix with size $max(n_l) \times (L-1)$ contain a collection of column vector θ_l, $\theta(1 : n_{l+1}, l)$denoting the bias vector from layer l to layer l+1 with size $n_{l+1} \times 1$.. All unused matrix elements are padded with 0.
γ: a vector with size $n_{max(L)}$ containing the current used class vector.

Output:
W_{new}:the updated 3d array with size $max(n_l) \times max(n_l) \times (L-1)$ containing a collection of matrix W_l, $W(1 : n_{l+1}, 1 : n_l, l)$ denoting the combination weights from layer l to layer l+1 with size $n_{l+1} \times n_l$. All unused array elements are padded with 0.
θ_{new}: a updated matrix with size $max(n_l) \times (L-1)$ contain a collection of column vector θ_l, $\theta(1 : n_{l+1}, l)$denoting the bias vector from layer l to layer l+1 with size $n_{l+1} \times 1$.. All unused matrix elements are padded with 0.

B. Functional Description for MATLAB Scripts

1) Data Pre-processing: The following MATLAB scripts pre-process the image format data into the desired feature vector and class vector. Data is stored column-wise into .cvs format

$data\_formating\_hwchar\_uppercase.m$

This script pre-process the handwritten data set for uppercase characters. M = 64, Q = 26.

$data\_formating\_digit.m$

This script pre-process the handwritten data set for digits from 0 - 9. M = 64, Q = 10.

$data\_formating\_char\_uppercase.m$

This script pre-process the "natural scenes" data set for uppercase characters. M = 64, Q = 26.

$data\_formating\_char\_fnt.m$

This script pre-process the computer generated font data set for uppercase characters. M = 64, Q = 26.

2) Training and Testing: The following MATLABscripts first separate a fraction of data as testing data and use the rest as training data. Then it performs stochastic-gradient backpropagation algorithm to train the network and testing on the testing data. The activation function used is the softplus function. The performance on the testing data is measured as $1 - R_{test}$ which is the rate of correct recognition on the testing data.

$traning\_with\_testing\_sp\_ucchar.m$

This script performs training and testing on "natural scenes" uppercase characters using stochastic-gradient training and softmax activation function.

$traning\_with\_testing\_sp\_hwchar.m$

This script performs training and testing on handwritten uppercase characters using stochastic-gradient training and softmax activation function.

$traning\_with\_testing\_sp\_fnt.m$

This script performs training and testing on computer generated uppercase font characters using stochastic-gradient training and softmax activation function.

$traning\_with\_testing\_sp\_digit.m$

This script performs training and testing on handwritten digits using stochastic-gradient training and softmax activation function.

$traning\_with\_testing\_sm\_ucchar.m$

This script performs training and testing on "natural scenes" uppercase characters using softmax training.

$traning\_with\_testing\_sm\_hwchar.m$

This script performs training and testing on handwritten uppercase characters using softmax training.

$traning\_with\_testing\_sm\_fnt.m$

This script performs training and testing on computer generated uppercase font characters using softmax training.

$traning\_with\_testing\_sm\_digit.m$

This script performs training and testing on handwritten digits using softmax training.

$traning\_with\_testing\_ce\_ucchar.m$

This script performs training and testing on "natural scenes" uppercase characters using cross-entropy training.

$traning\_with\_testing\_ce\_hwchar.m$

This script performs training and testing on handwritten uppercase characters using cross-entropy training.

$traning\_with\_testing\_ce\_fnt.m$

This script performs training and testing on computer generated uppercase font characters using cross-entropy training.

$traning\_with\_testing\_ce\_digit.m$

This script performs training and testing on handwritten digits using cross-entropy training.

3) Testing and Output: The following MATLAB scripts run after $training\_with\_testing\_*\_*.m$ and will use its result to generate stem plots showing the rate of correct recognition for each uppercase character or digit.

$real\_testing\_sp\_ucchar.m$

This script generates the resulting stem plot for "natural scenes" uppercase characters testing data and neural network trained by stochastic-gradient training and softmax activation function.

$real1\_testing\_sp\_hwchar.m$

This script generates the resulting stem plot for handwritten uppercase characters testing data and neural network trained by stochastic-gradient training and softmax activation function.

$real\_testing\_sp\_fnt.m$

This script generates the resulting stem plot for computer generated uppercase font characters testing data and neural network trained by stochastic-gradient training and softmax activation function.

$real\_testing\_sp\_digit.m$

This script generates the resulting stem plot for handwritten digits testing data and neural network trained by stochastic-gradient training and softmax activation function.

$real\_testing\_sm\_ucchar.m$

This script generates the resulting stem plot for "natural scenes" uppercase characters testing data and neural network trained by softmax training.

$real1\_testing\_sm\_hwchar.m$

This script generates the resulting stem plot for handwritten uppercase characters testing data and neural network trained by softmax training.

$real\_testing\_sm\_fnt.m$

This script generates the resulting stem plot for computer generated uppercase font characters testing data and neural network trained by softmax training.

$real\_testing\_sm\_digit.m$

This script generates the resulting stem plot for handwritten digits testing data and neural network trained by softmax training.

$real\_testing\_ce\_ucchar.m$

This script generates the resulting stem plot for "natural scenes" uppercase characters testing data and neural network trained by cross-entropy training.

$real1\_testing\_ce\_hwchar.m$

This script generates the resulting stem plot for handwritten uppercase characters testing data and neural network trained by cross-entropy training.

$real\_testing\_ce\_fnt.m$

This script generates the resulting stem plot for computer generated uppercase font characters testing data and neural network trained by cross-entropy training.

$real\_testing\_ce\_digit.m$

This script generates the resulting stem plot for handwritten digits testing data and neural network trained by cross-entropy training.

4) Parameter Optimazation: The follwing MATLAB scripts generate heatmaps for the performance $1 - R_{test}$ regarding to different step size μ and regularization factor ρ in logspace.

$training\_test\_param\_optimazation\_sigmoid.m$

This script generates the heatmap using stochastic-gradient training with sigmoid activation and the "natural scenes" uppercase characters data.

$training\_test\_param\_optimazation\_sp\_ucchar.m$

This script generates the heatmap using stochastic-gradient training with softplus activation and the "natural scenes" uppercase characters data.

$training\_test\_param\_optimazation\_sp\_hwchar.m$

This script generates the heatmap using stochastic-gradient training with softplus activation and the hand-

written uppercase characters data.

$training\_test\_param\_optimazation\_sp\_fnt.m$

This script generates the heatmap using stochastic-gradient training with softplus activation and the computer generated uppercase characters data.

$training\_test\_param\_optimazation\_sp\_digit.m$

This script generates the heatmap using stochastic-gradient training with softplus activation and the handwritten digit data.

$training\_test\_param\_optimazation\_sm\_ucchar.m$

This script generates the heatmap using softmax training and the ”natural scenes” uppercase characters data.

$training\_test\_param\_optimazation\_sm\_hwchar.m$

This script generates the heatmap using softmax training and the handwritten uppercase characters data.

$training\_test\_param\_optimazation\_sm\_fnt.m$

This script generates the heatmap using softmax training and the computer generated uppercase characters data.

$training\_test\_param\_optimazation\_sm\_digit.m$

This script generates the heatmap using softmax training and the handwritten digit data.

$training\_test\_param\_optimazation\_ce\_ucchar.m$

This script generates the heatmap using cross-entropy training and the ”natural scenes” uppercase characters data.

$training\_test\_param\_optimazation\_ce\_hwchar.m$

This script generates the heatmap using cross-entropy training and the handwritten uppercase characters data.

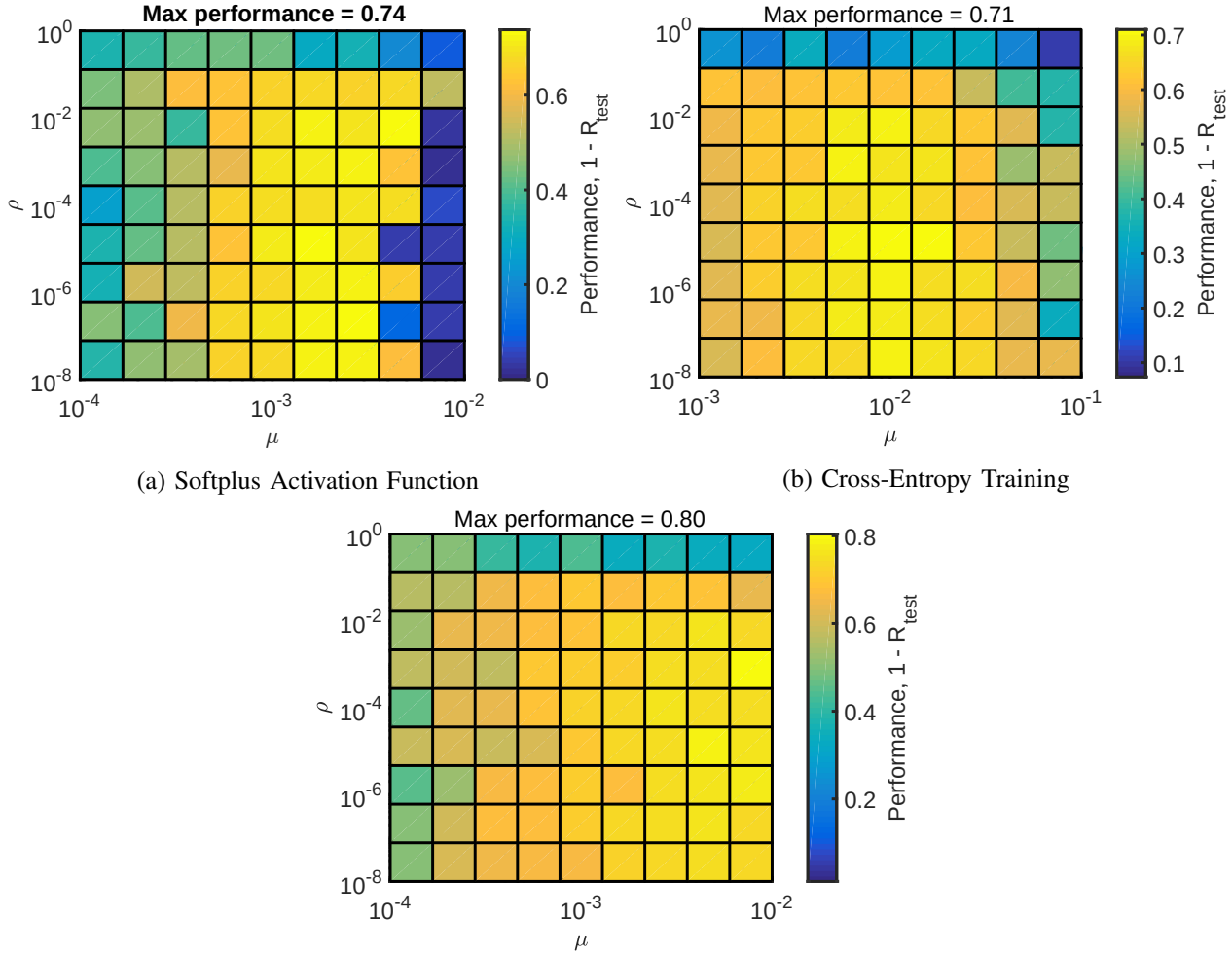
$training\_test\_param\_optimazation\_ce\_fnt.m$

This script generates the heatmap using cross-entropy training and the computer generated uppercase characters data.

$training\_test\_param\_optimazation\_ce\_digit.m$

This script generates the heatmap using cross-entropy training and the handwritten digit data.

C. Parameter Heatmap for The Other Three Datasets

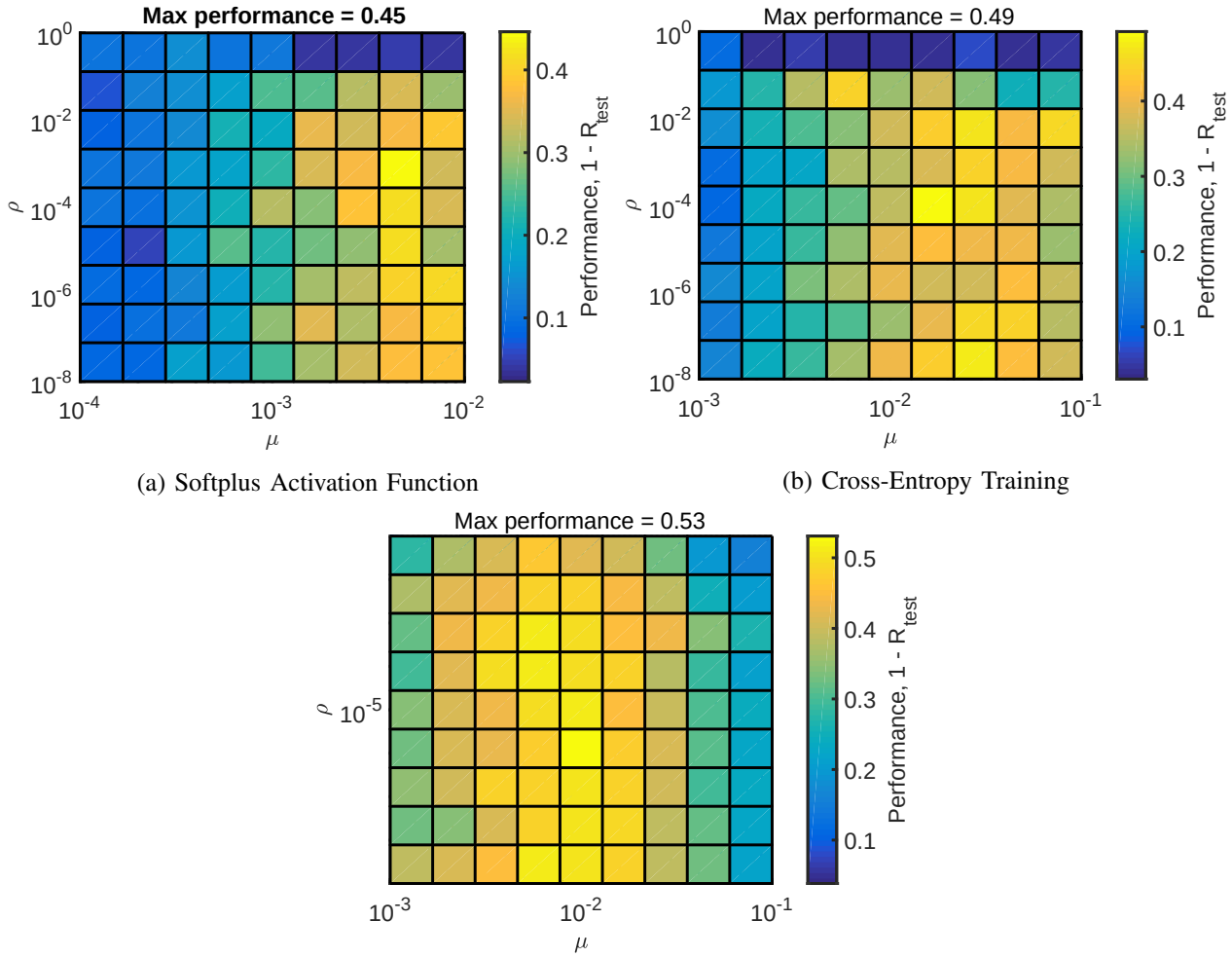


(a) Softplus Activation Function

(b) Cross-Entropy Training

(c) Softmax Training

Fig. 11: Parameter heatmap for natural scenes, for P=10

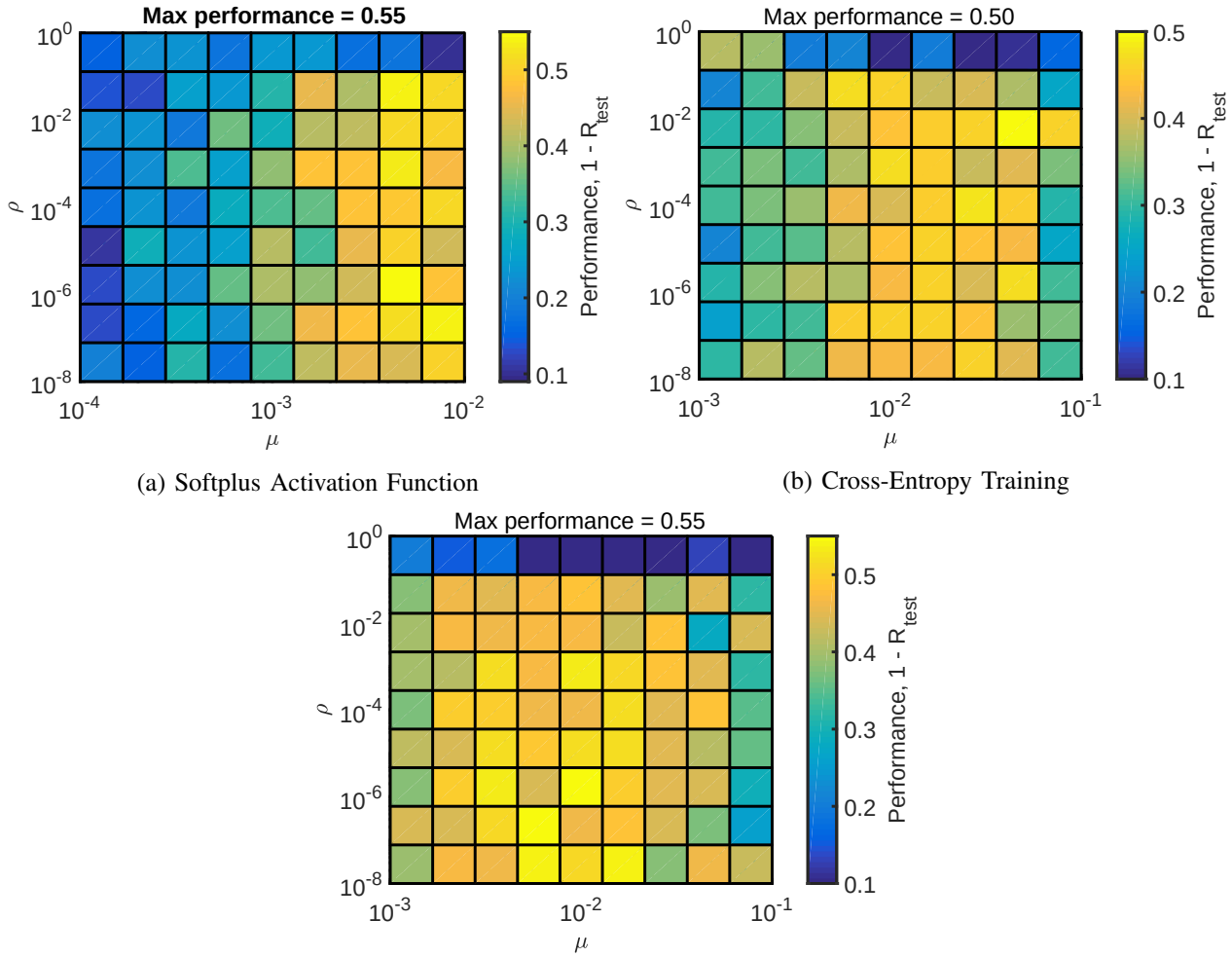


(a) Softplus Activation Function

(b) Cross-Entropy Training

(c) Softmax Training

Fig. 12: Parameter heatmap for handwritten characters, for P=10



(a) Softplus Activation Function

(b) Cross-Entropy Training

(c) Softmax Training

Fig. 13: Parameter heatmap for handwritten digits, for P=10