Neural Networks Project 3

Sagar Barbhaya, Jie Yuan, Robert McCartney

Executive Summary

While the MNIST data set of 10 handwritten digits has been successfully classified to levels below the error rate of humans, the limited number of classes makes it unclear whether this could be applied to a wider vocabulary of symbols. We have acquired three datasets each containing more than 100 characters from the Indian languages Devnagari, Tamil, and Telugu, and hope to evaluate the accuracy of Convolutional Neural Networks (CNN) in classifying these characters. Additionally, parameters describing the architecture and training behavior of CNNs will be tested against the resulting accuracy of the models and the training time.

Requirements and Specification

The recognition of handwritten characters based on pixel data is a popular application of current research in neural networks. One of the most well-studied data sets from this field is the MNIST digit set, which includes a sets of handwritten digits ‘0’-‘9’, on which state-of-the-art neural network-based models have achieved sub-1% accuracies, well below the error rate for human classification.[1][2] Despite this high performance, the ability to correctly classify 10 classes is rather limited compared to the diversity of visual stimuli that human brains can recognize. To test a broader range of characters, we have obtained three data sets of characters from Indian languages: Devanagari, Tamil, and Telugu, each of which contains more than 100 characters, far more than what are commonly used in English language text. Thus correctly identifying these characters is a much more challenging task.

Previously, traditional Multilayered Perceptron (MPL) and Radial Basis Function (RBF) neural networks were trained and evaluated on subsets of the three Indian language data sets: the MLP achieved the best results, at about 99.9%, 95.9%, and 89.6% for test subsets of the Devnagari, Tamil, and Telugu data sets, respectively. The RBF achieved accuracies of 85.2%, 84.3%, and 86.5%, respectively. However, in recent years more state-of-the art methods have been developed to process image-based data. Specifically, we would like to investigate the performance of a Convolutional Neural Network (CNN), which can process overlapping regions of input images similar to rod and cone cells in biological eyes. CNNs have already achieved high performance on the MNIST data set, but we would like to examine how well this performance scales to the classification of more than 100 classes simultaneously.

The CNN used for this study is the cuda-ConvNet package for Matlab, which includes functionality for gpu-based parallel computation.[3] All experiments were run on a GeForce GTX 960 GPU, with 1024 CUDA cores and 2 GB memory. In addition, NVIDIA’s CUDA Deep Neural Network library (cuDNN) was installed as the convolution primitives inside the MatConvNet library. In addition to testing model accuracy using the CNN, a number of parameters used to tune the performance of the CNN will also be varied over a range of values and judged against the performance. Once optimal parameters are discovered, these will be combined to test whether they result in a model with superior classification accuracy. As we already achieved high accuracies for the three Indian data sets using a simple MLP across the input pixels, we predict that the CNN will yield high accuracies overall. However, since the MLP performed with a significantly lower accuracy of 89.6% on the Telugu data set, we may focus most of our attention on improving the accuracy of this particular data set. Since the Telugu data set is also the smallest data set of the three, the lower performance is likely due to the smaller amount of available training data.

Feasibility and Analysis

We approached this project with an interest in exploring some of the recent developments in neural network research, including CNNs and deep learning for image-based recognition. CNNs are especially interesting because they were designed to mimic how actual eyes work: the network comprises pairs of two types of layers that alternate as the signal is passed from the input to the output layer:

1. Convolutional layers involve applying a trained kernel (or image mask) over the input image as a matrix of pixels, resulting in a new image matrix of the same size as the input image, in which each pixel is a weighted sum of its surrounding pixels after applying the weights specified in the kernel. This design is analogous to the function of rod and cone cells in the retinas of real eyes: identical sets of these cells are placed side-by-side in a grid around the curved retina, so that light entering the eye from different directions will trigger different sets of these cells. The convolutional kernel represents the receptive field of one of these cells, which is then applied to every region of the image. Applying the same kernel over the entire image allows for the detection process to be more robust to translational changes in the training data – for example, the desired symbol may be present in some image, but shifted to the left or right. Then the output pixels at the original location may not recognize the symbol, but the kernel will still be triggered at the shifted coordinates to the left or right. Also, in these layers multiple types of kernels can be applied to the image simultaneously and have their outputs fed together into the next layer.
2. Subsampling layers also involve applying a kernel to the input image, the convolved output of the previous layer, but the kernel function in this case is typically a max or average function. Also, the input region is mapped to only a single pixel in the output image at the location of the input region's center pixel, resulting in an image of reduced size. This is analogous to cells in retinas that have connections to regions of rod and cone cells – activations in any of the included cells will activate this connector cell, which transmits its signal toward the brain. The max function achieves this as well – an activation in any one of the convolutional cells in the kernel encompassed by a given subsampling cell will simply be retransmitted by the subsampling cell, unless a stronger

signal was already seen elsewhere. This design builds on the translational invariance property given by applying the same kernel at every location in the previous convolutional layer: instead of checking only a single set of pixels for a signal, the subsampling layer checks a region of convolutional locations, so the target symbol will be correctly identified in any area within that region.

3. After several stages of convolutions followed by subsampling, the outputs of the final subsampling layer are fed as input into a traditional multi-layered perceptron, comprising a hidden and output layer. The output layer corresponds to a vector representing the set of classes as its components: for example, for a set of 10 classes, there will be 10 output nodes, and the classification is assigned as the output node with the largest magnitude.

These changes to the traditional MLP architecture allow CNNs to more accurately learn relevant features in image data. We have chosen to use the cuda-ConvNet package developed by Krizhevsky et al. [cite], because it is written in Matlab, which makes image and matrix manipulation very convenient, and it is a very developed library with many modifiable parameters, including being able to fully specify the number and types of the layers in the network to be trained. Additionally, it provides automatic functionality for gpu-acceleration provided that a graphics card is available on the machine.

Implementation

The cuda-ConvNet provides an example CNN used to train and evaluate the MNIST data set, and the parameters for this model were used as the default model for training the Indian data sets. The batch size, the number of training instances to process in one epoch (iteration) of the backpropagation algorithm, is set to 200. The number of epochs is set to 150. The learning rate is set to 0.001, and the weight decay of 0.0005 (used to reduce the contribution of weights that are not frequently activated), and a momentum fraction of 0.90. This means that the weight update at a given iteration is not simply the error calculated at that particular iteration, but a weighted sum of the previous weight update and the current one. This allows gradient descent to escape local minima by overshooting these minima based on the often higher-gradient contribution of the previous weight update.

The architecture of the default network includes three pairs of convolutional and subsampling layers, but the last subsampling layer is replaced by a Rectified Linear Unit (ReLU) layer, which replaces the traditional sigmoid activation function with a max function. The advantage of this is that over many iterations, the max function is much more computationally efficient to compute than the sigmoid or tanh activation functions. Though this no longer bounds the output to the range of [0,1] or [-1,1], a normalization function can be applied to shift the set of values back into the desired range. In this case, following the ReLU layer, a final convolutional layer is applied, and finally, a softmax loss function is applied to generate the output. The softmax function is used to normalize a set of exponentially-scaled values to the range [0,1], and it is used in this case because the inputs into the layer have been log-transformed.

Experiments

The primary experiment involves splitting each of the Indian data sets into 70/30 training and test sets, training a CNN model on the training set, and then evaluating the classification accuracy when applying the test set. If the model does not perform at a high accuracy for any of the data sets, then those data sets will be run again while changing one of several model parameters over a range, to find the optimal parameters.

Results

To confirm that our version of cuda-ConvNet was functioning, we trained the default model on the MNIST data set, which was pre-split into training and test sets. Using this model, in only around 25 epochs, the training set error decreased to 0, and the test set error reached its optimal performance of about 1% error, as shown in Figure 1. The confusion matrix for the 10 digits is show in Table 1, and it shows that no clear patterns of misclassification were present – any single error only had a single-digit number of occurrences.

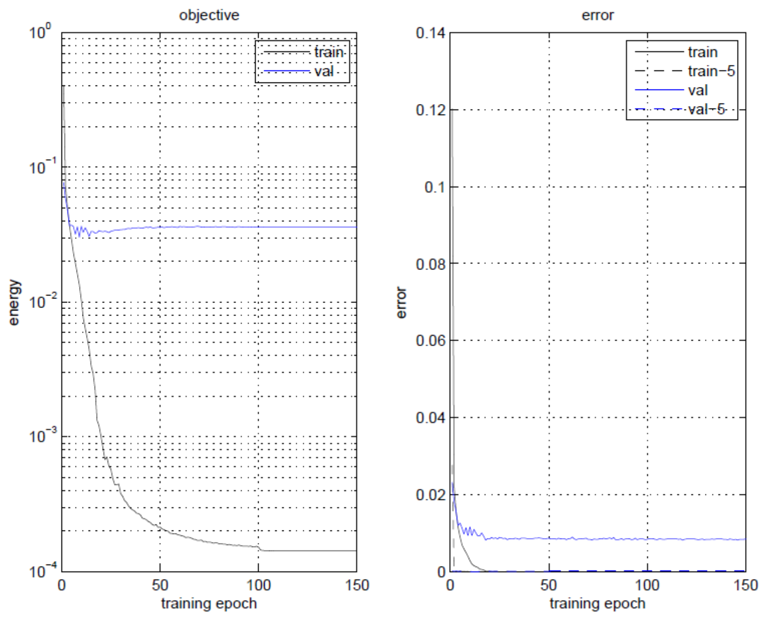


Figure 1: MNIST performance using default ConvNet model

	PREDICTED									
TRUE	0	1	2	3	4	5	6	7	8	9
0	975	0	1	1	0	2	4	0	2	0
1	0	1128	1	0	0	0	2	2	0	0
2	0	1	1026	1	1	0	0	3	1	0
3	0	1	0	1002	0	6	1	0	0	1
4	0	0	1	0	975	0	2	0	0	4
5	0	1	0	3	0	880	2	0	1	3
6	1	1	0	0	1	1	946	0	1	0
7	1	1	3	0	1	1	0	1020	1	3
8	3	2	0	3	0	1	1	1	966	1
9	0	0	0	0	4	1	0	2	2	997

Table 1: MNIST confusion matrix

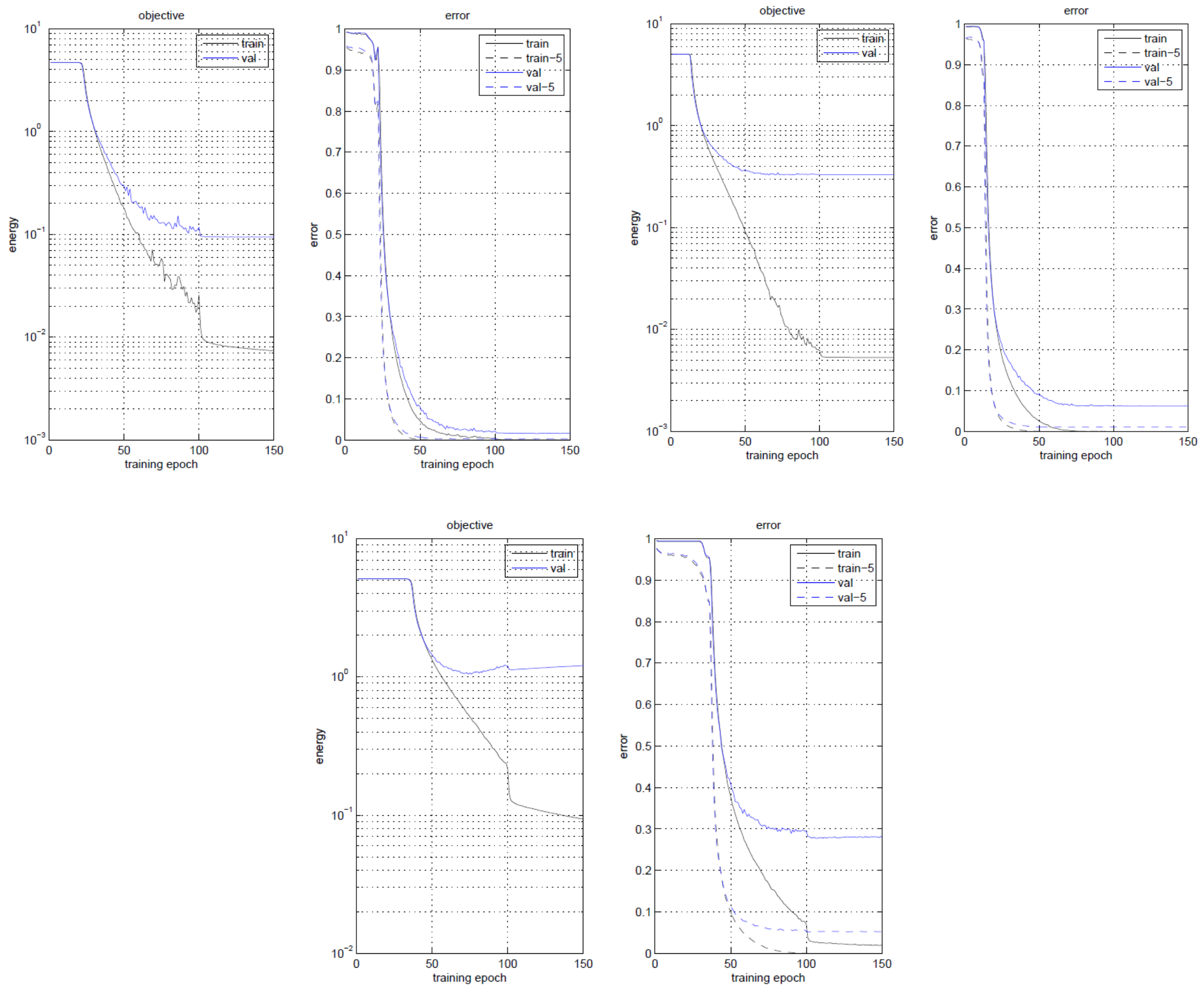


Figure 2: Training performance using default model: (Top Left) Devnagari; (Top Right) Tamil; (Bottom) Telugu. Val-5 and Train-5 refer to the best of the top 5 guesses by the model.

The three Indian data sets were then run using the same default model, yielding the results in Figure 2. Regarding the overall accuracy, the Devnagari data set reached an optimal accuracy of 98.4% at 75 epochs; the Tamil data set reached an optimal accuracy of 98.9% at around 60 epochs; and the Telugu data set reached only an optimal accuracy of 72.0% at around 100 epochs. Though the Devnagari and Tamil data sets achieved similar high accuracies as the MNIST data set, despite having significantly more character classes, the Telugu data set performed markedly worse based on the performance plots in Figure 2. In addition, the training performance appears to have been rather unstable – unlike the relatively smooth gradient descent processes of the other data sets, there is a sharp jump to a local error minimum at 100 epochs, after which the model is unable to improve any further. The Telugu data set is also by far the smallest of the three, containing approximately the same number of character classes but fewer training instances per character on average than the Devnagari and Tamil data sets, so it is possible that this relative lack of training information is responsible for the unstable and poorer-performing model. Figure 3 displays heat maps representing the confusion matrices for the evaluation of each of the three Indian data sets. By visualizing the errors, it is apparent that the Devnagari data set has very few errors distributed sporadically through the set, but the Tamil and Telugu sets contain certain errors that occur more frequently. And while the Tamil data set contains many more errors which individually occur relatively infrequently, the Telugu data set, which has the worst overall performance, appears to very frequently misclassify a small cluster of characters which appear prominently in the confusion matrix. Overall, however, the performance of the Telugu data set is reasonable given the large number of symbol classes (169). In cases with many classes, the criteria for successful classification is sometimes relaxed to allow for the true symbol class to be among the top 5 selections by the model. If this criterion is used, then the accuracy using the Telugu data set drops to under 5%.

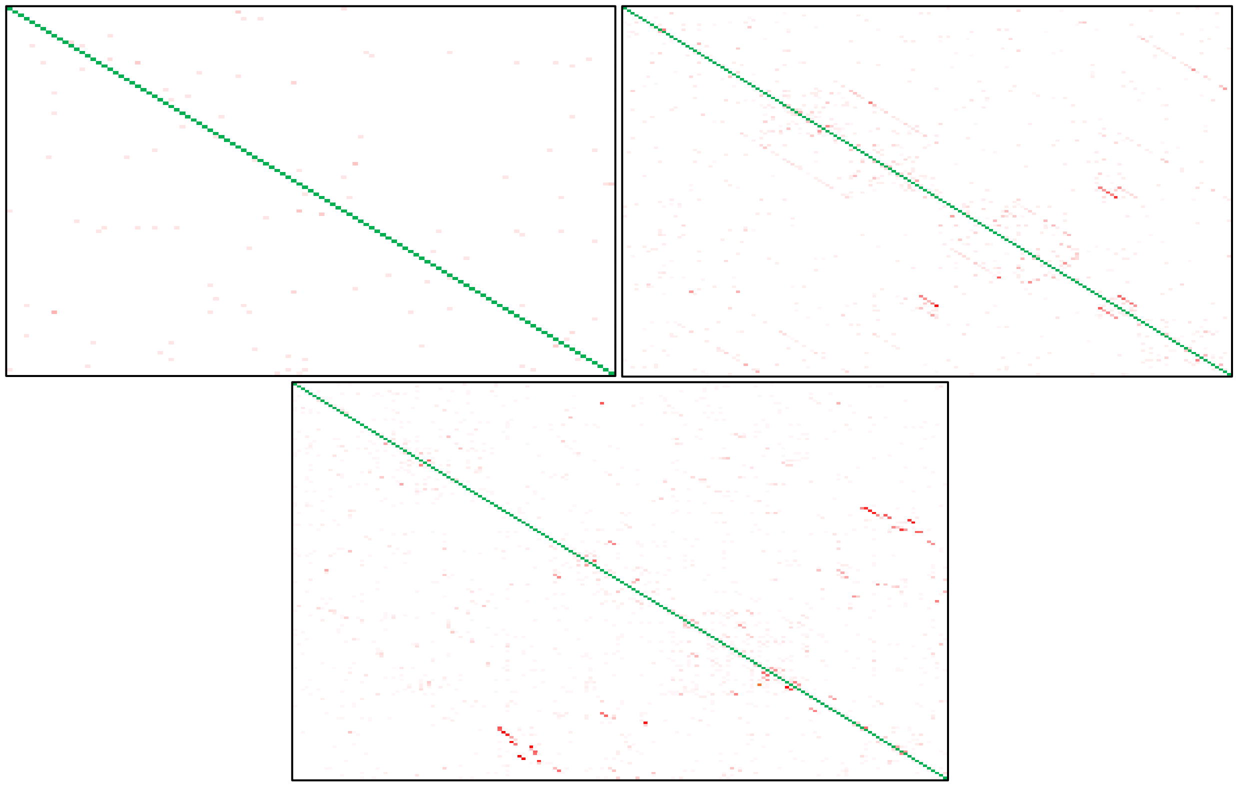


Figure 3: Confusion heat maps for Devnagari (Top Left), Tamil (Top Right), and Telugu (Bottom). Green indicates correct matches; red indicates errors.

Because the Devnagari and Tamil data sets already achieve a high classification rate using the default model, we believed that further refinement of this model would yield only slight improvements which would be difficult to analyze comparatively. Thus we focused most of our model refinement on improving the performance of the Telugu data set. For each model parameter that we varied, both the resulting accuracy using the new model, and the time required to train the model are recorded. The training time is determined as a rough estimate based on the training error plots, and is interpreted as the training epoch when the single validation error levels out and no longer improves.

	training epochs	Accuracy %
Momentum		
0.99	50	76.9
0.5	50	76.7
0	50	76.4
Dropout		
0.5	60	80.4
0.75, 0.5	80	76.9
0.75, 0.5, 0.5	7	0.5
Batch Size		
200	50	76.5
1000	25	77.7
4000	30	72.6
Topology		
0 conv. Layers	30	70.2
1 conv. Layer	50	78.9
3 conv. Layers	3	0.6

Table 2: Telugu results after varying CNN model parameters.

One of the parameters varied was the momentum – a high momentum value indicates that each new gradient that is calculated at a given iteration plays a very small role in adjusting the current moving average gradient, whereas a momentum value of 0 indicates that the gradient used to update the weights is simply taken as the new gradient at each iteration, and a moving average is not calculated. Based on the results of three momentum values shown in Table 2, the momentum value does not appear to significantly influence both the recognition rate and the training time.

Another tested parameter is the batch size, which specifies the number of samples to perform the error calculation and weight update on during each epoch. The more samples run at once, the faster the convergence will tend to be, since more information is incorporated per epoch, and this is what is seen generally in the results in Table 2. However, the resulting accuracies did not improve appreciably and even decreased with a very large batch size – this could be because with small batch sizes, some randomness is introduced by which examples are included in each batch. But the effect of this randomness is decreased once a majority of the data is used at each iteration, so the increase in error could be the model failing to fully explore subregions of the error space.

Also, the addition of dropout layers was investigated – these are additional layers added after the convolutional layers which are modified before each training iteration: each of the connections between the previous layer and the dropout layer has a 50% chance of being inactivated, so the output of the previous layer is not transmitted to the dropout nodes. Adding a single layer with a 50% chance of dropping each node generated a significant increase in the model's accuracy, from 72.0% to 80.4%. The increase in randomness added by the dropout process is also visible in the training process shown in Figure 4: for around the first 30 epochs, the training error fluctuates rapidly, indicating that the random dropout is causing the model to sample from many different areas of the weight space. However, when additional dropout layers are added, including a layer in which each node has a 75% chance of being dropped, the error increases dramatically – with three layers, the accuracy is only 0.5%. In this case, every character got classified as a single character class, so it is likely that at some point in the training process too many weights were dropped, which caused the model to enter a suboptimal error minimum. Additionally, there are likely too few training examples to fully train a model with the added complexity of 3 dropout layers. The plot detailing this unstable behavior is show in Figure 5.

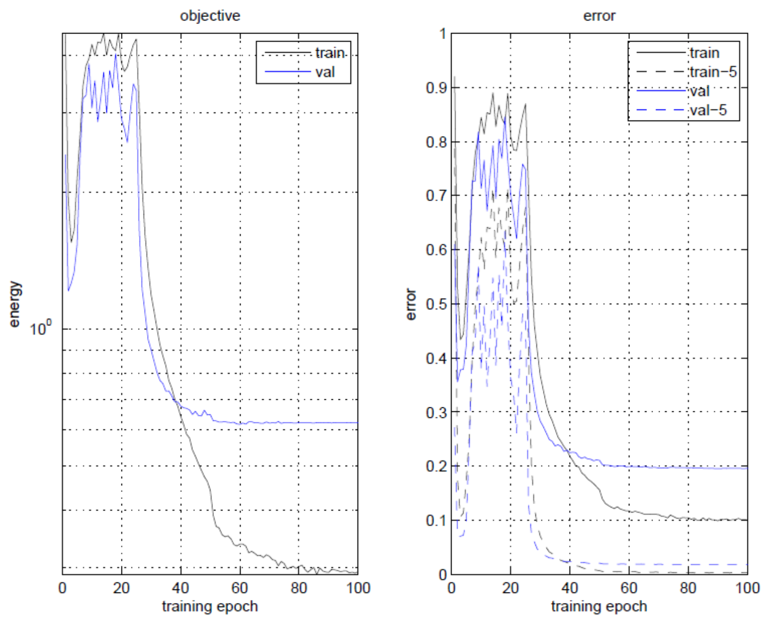


Figure 4: Error plot with one dropout layer. Early training is erratic, but the model enters a slightly improved error minimum.

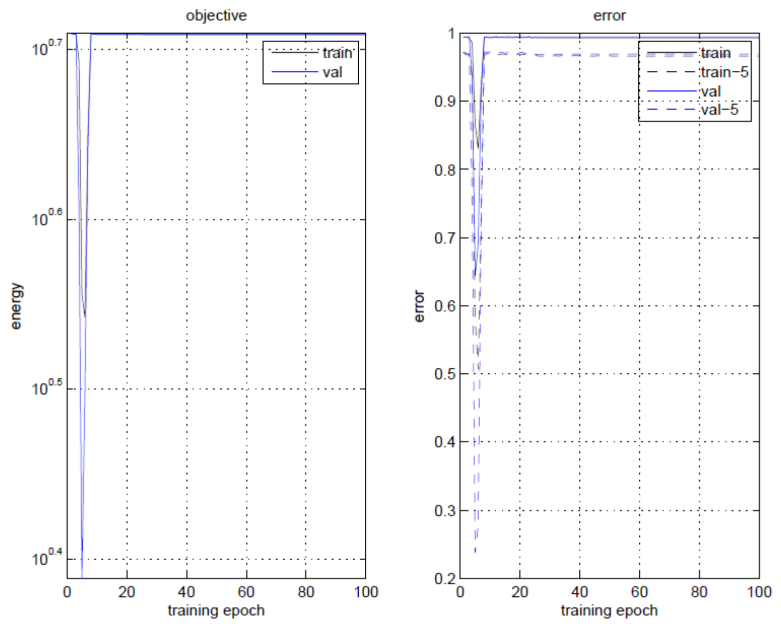
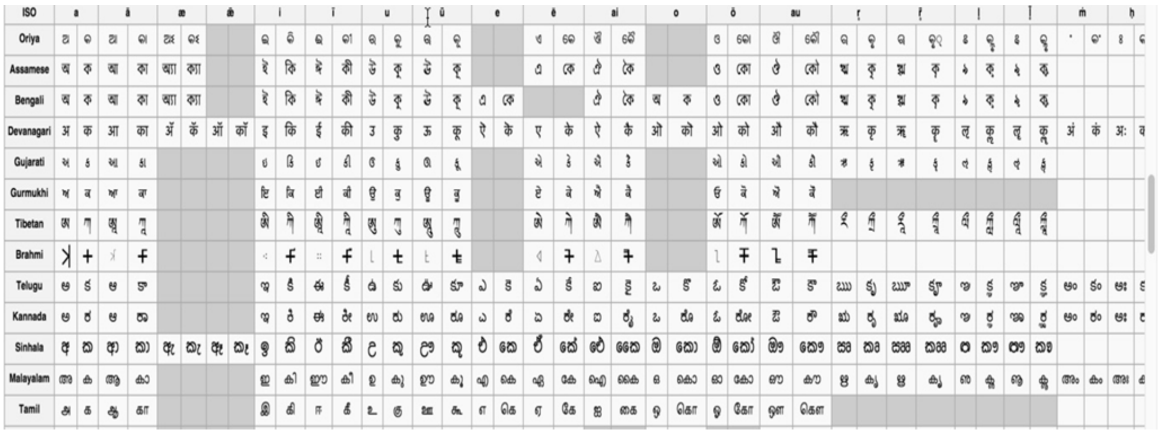


Figure 5: Error plot with three dropout layers. Too many layers and too few data causes the model to enter a suboptimal minimum.

Finally, the topology of the model was investigated by varying the number of convolutional layers that are computed at the early stages of the model. With no convolutional layers, the model becomes a simple MLP neural network, and surprisingly, the accuracy of this model (70.2%) is only slightly worse than the original CNN model. Surprisingly, using a single convolutional layer slightly outperforms the default model, which uses two. And adding yet another convolutional layer caused the model to fail to classify altogether, as it again assigned every character to a single class and achieved a sub-1% accuracy. As in the previous case where too many dropout layers were added, it seems that there is not enough data to fully train the model once it starts to increase past a certain point of complexity with the addition of many layers, so the result is a model whose weights have not been adequately tuned to represent the boundaries between the many character classes.

Conclusion

The poor performance on the Telugu data set means that the characters could not be identified very accurately. Why did that happen lesser with the Tamil dataset? Basically, all three languages, Tamil, Telugu and Devanagari are “Brahmic languages”. All are derived from a language called “Sanskrit”. Just like English has the 26 letters, there are a set of predefined letters in these languages. The way these are written in each of these languages differs though! Here is hidden one of the reasons that we are trying to find.



ISO	a		ā		æ		ǣ		i		ī		u		ū		e		ē		ai		o		ō		au		ṛ		ṝ		ḷ		ḹ		ṁ		ḥ	
Oriya	ଅ	କ	ଆ	କା	ଅ୍ୟ	କ୍ୟ			ଇ	କି	ଈ	କୀ	ଉ	କୁ	ଊ	କୂ			ଏ	କେ	ଐ	କୈ			ଓ	କୋ	ଔ	କୌ	ଋ	କୃ	ୠ	କୄ	ଌ	କୢ	ୡ	କୣ	ଂ	କଂ	ଃ	କ
Assamese	অ	ক	আ	কা	অ্যা	ক্যা			ই	কি	ঈ	কী	উ	কু	ঊ	কূ			এ	কে	ঐ	কৈ			ও	কো	ঔ	কৌ	ঋ	কৃ	ৠ	কৄ	ঌ	কৢ	ৡ	কৣ				
Bengali	অ	ক	আ	কা	অ্যা	ক্যা			ই	কি	ঈ	কী	উ	কু	ঊ	কূ	এ	কে			ঐ	কৈ	অ	ক	ও	কো	ঔ	কৌ	ঋ	কৃ	ৠ	কৄ	ঌ	কৢ	ৡ	কৣ				
Devanagari	अ	क	आ	का	अँ	कँ	ऑ	कॉ	इ	कि	ई	की	उ	कु	ऊ	कू	ऎ	कॆ	ए	के	ऐ	कै	ऒ	कॊ	ओ	को	औ	कौ	ऋ	कृ	ॠ	कॄ	ऌ	कॢ	ॡ	कॣ	अं	कं	अः	क
Gujarati	અ	ક	આ	કા					ઇ	કિ	ઈ	કી	ઉ	કુ	ઊ	કૂ			એ	કે	ઐ	કૈ			ઓ	કો	ઔ	કૌ	ઋ	કૃ	ૠ	કૄ	ઌ	કૢ	ૡ	કૣ				
Gurmukhi	ਅ	ਕ	ਆ	ਕਾ					ਇ	ਕਿ	ਈ	ਕੀ	ਉ	ਕੁ	ਊ	ਕੂ			ਏ	ਕੇ	ਐ	ਕੈ			ਓ	ਕੋ	ਔ	ਕੌ												
Tibetan	ཨ	ཀ	ཨཱ	ཀཱ					ཨི	ཀི	ཨཱི	ཀཱི	ཨུ	ཀུ	ཨཱུ	ཀཱུ			ཨེ	ཀེ	ཨཻ	ཀཻ			ཨོ	ཀོ	ཨཽ	ཀཽ	རྀ	ཀྲྀ	རཱྀ	ཀྲཱྀ	ལྀ	ཀླྀ	ལཱྀ	ཀླཱྀ				
Brahmi	𑀅	𑀓	𑀆	𑀓𑀸					𑀇	𑀓𑀺	𑀈	𑀓𑀻	𑀉	𑀓𑀼	𑀊	𑀓𑀽			𑀏	𑀓𑁂	𑀐	𑀓𑁃			𑀑	𑀓𑁄	𑀒	𑀓𑁅												
Telugu	అ	క	ఆ	కా					ఇ	కి	ఈ	కీ	ఉ	కు	ఊ	కూ	ఎ	కె	ఏ	కే	ఐ	కై	ఒ	కొ	ఓ	కో	ఔ	కౌ	ఋ	కృ	ౠ	కౄ	ఌ	కౢ	ౡ	కౣ	అం	కం	అః	క
Kannada	ಅ	ಕ	ಆ	ಕಾ					ಇ	ಕಿ	ಈ	ಕೀ	ಉ	ಕು	ಊ	ಕೂ	ಎ	ಕೆ	ಏ	ಕೇ	ಐ	ಕೈ	ಒ	ಕೊ	ಓ	ಕೋ	ಔ	ಕೌ	ಋ	ಕೃ	ೠ	ಕೄ	ಌ	ಕೢ	ೡ	ಕೣ	ಅಂ	ಕಂ	ಅಃ	ಕ
Sinhala	අ	ක	ආ	කා	ඇ	කැ	ඈ	කෑ	ඉ	කි	ඊ	කී	උ	කු	ඌ	කූ	එ	කෙ	ඒ	කේ	ඓ	කෛ	ඔ	කො	ඕ	කෝ	ඖ	කෞ	ඍ	කෘ	ඎ	කෲ	ඏ	කෟ	ඐ	කෳ				
Malayalam	അ	ക	ആ	കാ					ഇ	കി	ഈ	കീ	ഉ	കു	ഊ	കൂ	എ	കെ	ഏ	കേ	ഐ	കൈ	ഒ	കൊ	ഓ	കോ	ഔ	കൌ	ഋ	കൃ	ൠ	കൄ	ഌ	കൢ	ൡ	കൣ	അം	കം	അഃ	ക
Tamil	அ	க	ஆ	கா					இ	கி	ஈ	கீ	உ	கு	ஊ	கூ	எ	கெ	ஏ	கே	ஐ	கை	ஒ	கொ	ஓ	கோ	ஔ	கௌ												

Table 3: Variation of letters across various Brahmic languages.

In Table 3, the columns are the different letters like consonants in English. Each of the rows represents different Brahmic languages. If we carefully look at the Tamil, Telugu and Devanagari languages, we can see that Tamil has a lot more missing letters than the other languages, this is because, in Sanskrit, some letters are different from each other on the way they are pronounced (optional letters), so Tamil may not have all of these incorporated in itself. How does this help us analyze our results?

As the number of values to be compared to reduces, the process definitely becomes more efficient. That is exactly what seems to have happened with the Tamil dataset. Also, one more thing which can be one of the reasons leading to poor performance could be the way these letters are. If we see carefully, the Telugu consonants are much similar looking to each other than the Tamil. So it becomes even tougher with greater number of consonants, which are similar looking, available to be compared to.

All these reasons also lead to finding out where exactly in the code were things failing or not allowing us to give accurate results all the time. There were 3 files (names might differ a but depending on improvements that the group has been making but I shall explain exactly where it is in the code). There are a combination of processes that happen back and forth within certain classes that get affected due to the reasons mentioned above. Let us see how. The classes getMnist, convert_normalize are the ones that are most susceptible considering the fact that the data for comparison and the comparison happens here. Now this is susceptible since the comparison is the one in which the more the data, the easier it is for the Tamil consonants to be compared considering that they are fewer in number than the Telugu or the Devanagiri.

Also, considering the train_back, main classes are the places where the characters are tried to be differentiated, after seeing how similar looking Telugu characters are this will become really difficult for that dataset compared to Tamil dataset. Now this explains why even after having implemented the same code and procedure for all the 3 datasets, we get different results for all 3 of them.

References

1. Y. LeCun, L. Bottou, Y. Bengio and P. Haffner: Gradient-Based Learning Applied to Document Recognition, Proceedings of the IEEE, 86(11):2278-2324, November 1998.
2. D. Ciresan, U. Meier, and J. Schmidhuber: Multi-column Deep Neural Networks for Image Classification, Computer Vision and Pattern Recognition, 3642-3649, February 2012.
3. Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton: Imagenet classification with deep convolutional neural networks, Advances in neural information processing systems, 1097-1105, 2012.