Project 3: Modified Digits
Team Name on Kaggle : DNS

David Newton
260597626
david.newton2@mail.mcgill.ca

Nirmal Kanagasabai
260716737
nirmal.kanagasabai@mail.mcgill.ca

Senjuti Kundu
260669284
senjuti.kundu@mail.mcgill.ca

1. Introduction

The goal of this project is to utilize three different learning algorithms to automatically classify images into one of forty different classes. The learning algorithms explored in the project are the following: logistic regression (LR); feed-forward neural networks (NN); and convolutional neural networks (CNN).

Image recognition is a fundamental problem that connects many disciplines from computer vision and robotics to medicine and astronomy. Initial research into the field started in the 1960s in artificial intelligence departments. This first work was influenced heavily by the study of biological precedents, such as human and animal vision systems. The goal of early computer vision research was the full reconstruction of of 3D scenes as well as extraction and understanding of that scene. In the 1970s the focus of the research shifted to developing efficient algorithms that could work with the technology of the time to extract images features like edges and corners. Popular computer vision algorithms like the "Harris Corner Detector" [1] and the "Canny Edge Detector" [2] were developed at this time. In the 1980s and 90s, statistical models were developed and applied to computer vision problems related to 3D reconstruction with important results.

Statistical methods dominated computer vision research until 2012 when researchers using Convolutional Neural Networks in the popular ImageNet competition bested all other approaches, achieving an error rate of 15.3% [3]. This compared to 26.2% by the next highest entry in the competition. This dramatic improvement in prediction accuracy has led to a surge of research in the area of neural networks that continues to the present and sets the context for the project described in this paper, in which three learning algorithms are explored that represent technologies available before and after the landmark competition described above. In the following sections this paper will compare and contrast the representation, implementation, training, and testing of logistic regression (LR), feed-forward neural networks (NN), and convolutional neural networks (CNN).

2. Problem Representation

The dataset consists of 26344 training images and 6600 testing images, all scaled down to 64x64 8-bit RGB images for manageability. Each training image has a unique label, between 0 to 39. The number of images per class is not equally distributed, Class 0 (the largest class) has 8000 images while Class 39 (the smallest class) has only 148 images.

2.1. Preprocessing the data

Traditionally, dimensionality reduction is conducted on images using PCA and images are converted to grayscale, before image classification can begin. However, the data presented to us was already very small in size (each image being only 64x64), so dimensionality reduction was not necessary. The RGB channel information was also important for feature extraction of the images and so they were not converted to grayscale either.

A key preprocessing step involved addressing the imbalanced number of examples that were available for each image category. In order to reduce any possible bias that an imbalanced data set might create, the team chose to use equal data samples from each image category for training. This meant that a potential bias could be avoided, but it also meant that some data would not be used.

2.2. Feature Selection

2.2.1. Logistic Regression. We have used the OpenCV Python implementation of SURF for feature selection.

SURF or "Speeded Up Robust Features" is an optimized algorithm for keypoint detection in an image.

Keypoints are spatial locations in an image which define interesting points in the image. They are invariant to image transformations (such as rotation, scaling) and distortions (homography). Each keypoint has an associated descriptor which is primarily concerned with the scale and orientation of the keypoint.

SURF keypoints and descriptors, once extracted from an image, are traditionally used to detect the same image later. This makes them suitable as features for classifying similar sets of images.

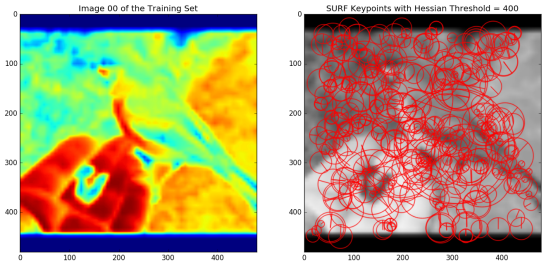

Figure 1. Comparison of the original image and the image with detected SURF keypoints.

Traditionally, the Hessian threshold is set to between 300-500 for feature extraction in SURF. Though a higher Hessian threshold enables quicker computation as less keypoints and descriptors are computed, it can also result in the case that no keypoints are computed for some images. Thus a balance needs to be struck. The greater the Hessian threshold, the fewer the number of keypoints identified. For our purposes, varying the Hessian from 1800 to 3000 enables to detect between 40 to 80 keypoints per image, which is adequate for our needs.

Other useful preprocessing steps that we could have pursued would have involved de-noising and Gaussian or Hessian smoothing.

2.2.2. Feed-Forward Neural Network. The input representation used for the implementation of the Neural Network for the project used raw pixel data without any SIFT or SURF feature extraction. Specifically, the 3 RGB 64x64 pixel arrays provided for each image was flattened into one large array consisting of 12,228 pixels per image. The developed NN then utilized all 12,228 pixels as inputs to the network. The RGB values of these inputs were then scaled to fit within a range of -1 and 1 - as recommended by some researchers.

The output/label values for each image were converted from a single integer value indicating the image category to a 40 element array with a 1 in the index representing the image category. The output data was also scaled to fit within the same -1 to 1 range as the input data. This representation of the output was an important transformation to fit with the NN architecture and allowed an image designation to be found by checking which of the array indexes was closest to 1.

2.2.3. Convolutional Neural Network.

3. Algorithms and Implementation

3.1. Logistic Regression

Logistic regression, though a linear model, has often been used for non-linear classification problems. We have used the scikit-learn library's implementation of L2 regularization in Python with multi-class Logistic Regression in order to prevent overfitting and enable automatic feature selection.

The cross entropy logistic loss function used to determine the weights, often does not converge to the optimal value. This has been observed in our case, owing to the large number of classes and training samples present.

When the model was fitted using the entire dataset, the results were highly skewed, with most images being classified as Class 0. This is expected, since Class 0 contains the largest number of images by far. Hence, we trained our model using the first 148 images of each class (because the smallest class has only 148 images), resulting in a total of 5920 training images. Another possibility we could have opted for would have been to select 148 images at random from each class for training, though we do not expect that this would have had an appreciable impact on classification accuracy.

Logistic regression is not typically used for image classification problems, which explains the low accuracy of our model.

3.2. Feed-Forward Neural Network

Feed-forward neural networks (NN's) are capable of representing a diverse array of linear and non-linear functions. NN's have also demonstrated a strength in image recognition problems and have become one of the dominant research areas in the field of computer vision in the form of deep NN's and CNN's. Some downsides to NN's include the problem of over-fitting and also the fact that a large amount of hyper-parameters must be optimized.

In order to classify a set of given images a custom NN for image recognition was developed and implemented for the project using the Python programming language. In addition to implementing the NN, a training class and cross-validation class was also developed and implemented to optimize key hyper-parameters involved in the NN. The NN was implemented to allow for mini-batch training with gradient descent. The implementation attempted to use matrix operations efficiently in order to speed-up feed-forward and backward-propagation tasks. Further, it should be noted that the activation function chosen for the project was a sigmoid function.

An important design choice in implementing the gradient descent portion of the NN algorithm was the choice of the stopping rule. The developed NN stopped the gradient descent iterations when the difference in the cost between the training and test results stopped decreasing. This meant that the algorithm could detect when over-fitting was beginning to happen.

Over-fitting is a common problem in NN's. To avoid over-fitting, the implemented NN utilized an L2 penalty regularization term as an addition to the cost function as seen in equation 1:

$$J(w) = 0.5(y - hw(x))^2 + 0.5\lambda \; wTw$$

3.3. Convolution Neural Networks

Convolution Neural Networks work by consecutively modeling small chunks of data and combining them deeper in network. They are more like a 'Black Box' which constructs features that we would otherwise have to handcraft. The abstract features that were created from training the model are generic. Hence, they take into account the variations of different test images. It typically consists of three types of layers: Convolution Layer, Pooling Layer and Fully Connected Layer. Typically, the first layer is used to perform edge detection. The subsequent layers include abstract operations like shape detection, tracking different object positions, illumination, scales, etc. The final layers make use of the templates created by the previous layers and match the input test image with the templates and predict a weighted sum of all of them. Hence, Convolution Neural Networks are able to model complex variations and behavior, producing highly accurate predictions.

Training a Convolution Neural Networks from scratch requires a lot of computing power and time. In this project, we employ a pre-trained Convolution Neural Network model called Inception. Inception was trained by Google on 100K images with 1000 categories. In order to adapt the pre-trained inception model to the Tiny Imagenet Challenge, we adopted the 'Transfer Learning' technique. As the name states, Transfer Learning is the process of applying knowledge obtained from a previous training session to a new training session.

The image recognition model (Inception-v3) comprises of a feature extraction module with Convolution Neural Network and a classification module with fully-connected and softmax layers. By using a pre-trained classifier like Inception-v3, we re-use the feature extraction module and re-train the classification part with our Tiny Imagenet dataset.

4. Training, Validation, and Testing Results

Each of the three algorithms (LR, NN, CNN) required their own training and validation approaches. In the following sections, each algorithm along with the details of its training and validation will be described.

4.1. Logistic Regression

Cross-validation was carried out using Python's scikit-learn package's implementation, with the training set divided as 80-20. Using the entire training image dataset of 26344 images yielded a low accuracy of between 31-35%, depending on the SURF Hessian threshold chosen.

4.2. Feed-Forward Neural Network

The large data set of images presented some key challenges for training. The primary challenge was computing time. To allow for quicker training, the image set was balanced and reduced so that for each image there were 148 images from each image category. This meant that 5,920 images were used for training to make final predictions. For cross-validation, a subset of 2,500 images containing equal numbers of images from each image category were used. This allowed the cross-validation testing to look at more values for the chosen hyper-parameters in less time.

The training involved using k-fold cross-validation to optimize the following hyper-parameters: the learning rate (or alpha value); the number of hidden layers in the network; and the number of hidden nodes on each hidden layer. The training of the NN was done with mini-batch gradient descent. This approach was chosen because it allowed the algorithm to efficiently use matrices to perform many operations at once on several samples, thereby saving computing time. Batch sizes of 50, 100, and 200 images were tried and smaller batch sizes seemed to perform better.

The results of the cross-validation can be seen in figures 2, 3 and 4. Figure 2 shows results for the cross-validation of the learning rate. For this test the number of hidden layers was set to 1 and the number of hidden layer nodes was set to half of the number of input nodes: input nodes = 12,228, so hidden layer nodes = 6,114. The results of the cross-validation showed that an alpha value of 0.5 worked best. This large alpha value did cause the exponential in the sigmoid function to overflow in the initial iterations however, triggering warnings in Python. The large error values in the figure (as well as figures 3, 4 and 5) are due to the fact that the cost of all the observations in the mini-batch are being summed together.

The results for the cross-validation of the number of hidden layers can be seen in figure 3. Here an alpha value of 0.5 is used and the number of hidden nodes is set to 1000. The figure shows a clear trend in which as more hidden layers are added (e.g. 1-3 layers), less aggregate error is seen. The opposite trend occurs as the number of nodes on the hidden layer is increased. Figure 4 clearly shows that as hidden nodes are added the error increases substantially.

Figure 5 shows a portion of the training in action. Specifically, it shows training vs. testing error for 2,500 images done with 2,00,000 mini-batch gradient descent iterations. The figure shows the relation of testing to training and does not depict the familiar over-fitting form in the graph. Further, many of the figures do not have this characteristic form. It seems likely that the reason for this might be in the mini-batch implementation, or perhaps due to the fact that more training iterations needed to be run to see this characteristic divergence between training and testing curves.

The training for the NN in figure 5 took about 5 hours and the predictions it produced for the test set were unfortunately very poor at 0.9% prediction accuracy. Several other attempts were made at retraining with equally bad results. It seems likely that the following issues might be to blame:

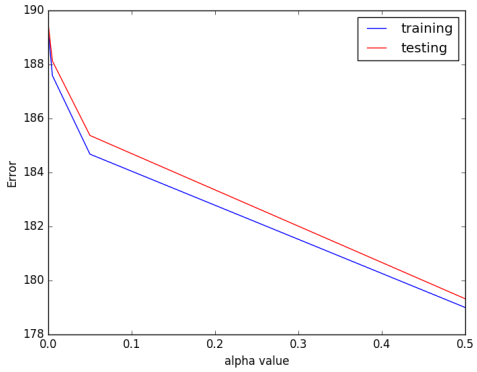

Figure 2. Cross-validation of the learning rate alpha.

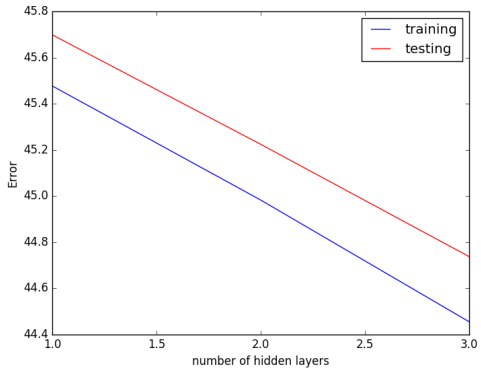

Figure 3. Cross-validation of the number of hidden layers.

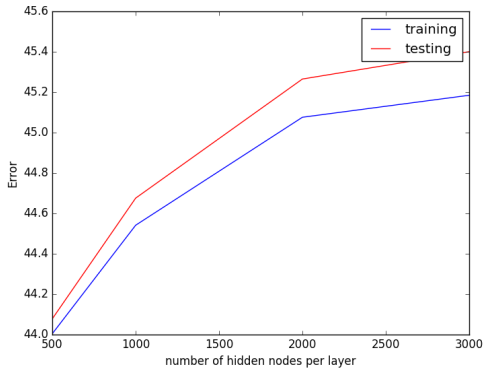

Figure 4. Cross-validation of the number of hidden layer nodes.

the choice to train on only 5,920 images might have been a problem; the use of the mini-batch gradient descent was problematic; there was an error in the code somewhere; or perhaps the selected hyper-parameters were bad.

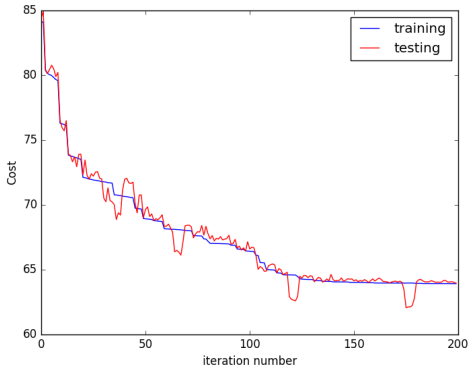

Figure 5. Training vs testing error for 2,500 images done with 200,000 mini-batch gradient descent iterations.

4.3. Convolution Neural Networks

In order to use the Transfer Learning approach, we had to extract the images from numpy arrays. Doing it on the fly wasn't a viable option as it took a long time and the process got very complex. Hence, we separated the Image extraction module and ensured that the image features were preserved. With the features being preserved in images, we had to ensure we have the labels of these images. The model needs to know what class each and every image belongs to. To do this, we pre-computed the number of unique classes and the count of images in each class using the testY.npy file, followed by image extraction to appropriate folders. The directories serve as the labels for the model.

We removed the old top layer of the pre-trained Inception v3 model and trained a new one with the Tiny Imagenet dataset provided. The magic of transfer learning was visualized when the lower layers that have been trained to distinguish between some objects can be reused for many recognition tasks without any alteration.

The layer just before the final layer of the model is called 'Bottleneck'. This penultimate layer has been trained to output a set of values which could be used by the classifier to distinguish between the classes it has been asked to recognize. Hence, it is very important and it must contain a meaningful and compact summary of the images. It must contain enough information for the classifier to make a good choice in a very small set of values. We created bottleneck values for all 26344 images after analyzing them. We also persisted the bottleneck values locally as it could be reused multiple times during our training and validation phase. Otherwise, it takes significant amount of time to run the classifier every time.

Once the bottleneck calculations are complete, the training of the final layer of the network begins. This process involved 4,000 training steps. In each step, 10 images were chosen at random from the training set, their bottlenecks were retrieved from the cache and fed into the final layer to get the predictions. These predictions were compared against the actual labels to update the final layer's weights through

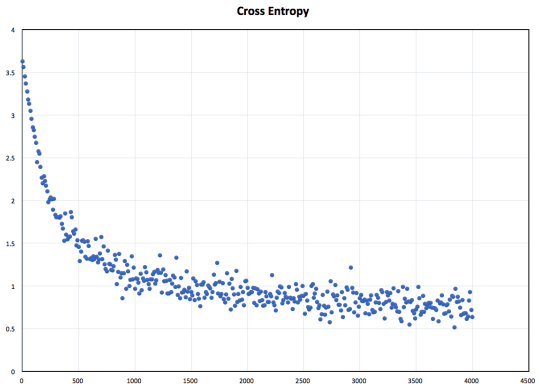

Figure 6. Cross Entropy.

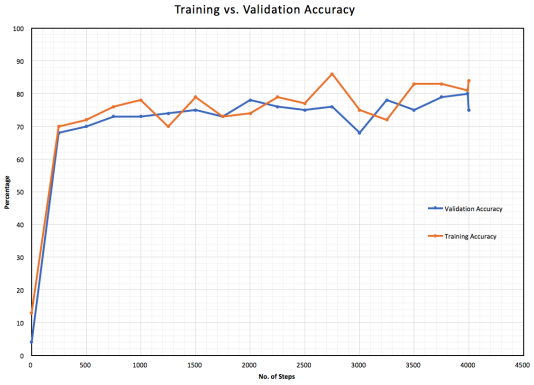

Figure 7. Cross-validation of the number of hidden layers.

the back-propagation process. We continued this process and the accuracy improved with successive iterations. We tried to increase the number of training steps from 4000 to 6000. However, it took twice as long and we had to kill the process mid-way due to lack of computation power.

The test evaluation process is the best estimate of how the trained model will perform on the classification task. The actual value will vary from one run to another as there is some amount of randomness in the training process. This number is based on the percentage of test images that are given the correct label after the model is fully trained.

We used the conventional 80-20 split on the training dataset so that we could perform cross-validation. Out of the 20, we further separated into 10 and 10 so that we could compute the final test accuracy of our model on the latter 10. We computed the training accuracy, validation accuracy and cross entropy in every step. The training accuracy depicts what percent of the images used in the current training dataset were labeled with the correct class. The validation accuracy is the precision on the 10 randomly selected images from the validation dataset. At points where the training accuracy is higher but the validation accuracy is lower in Fig. 7, we have over-fit the model where our classifier has memorized particular features in the training images that weren't very helpful. Cross Entropy is a loss function. It highlights the progress of our learner. As can be inferred from figure 6, the objective of our training is to make the loss as small as possible. We were able to get a final test accuracy of 78.5%. In Kaggle, our model performed better than the test we did internally. Our CNN classifier yielded an accuracy of 89.21%.

5. Discussion and Conclusion

The project involved the exploration of three learning algorithms to correctly classify images: Logistic regression (LR); neural networks (NN); and convolutional neural networks (CNN). The testing results, as presented in the previous sections, showed that the clear winner was the CNN. The CNN showcased a 89% accuracy, which far exceeded the results of the other two approaches. The LR and NN implementations both performed poorly. We expected LR to perform badly, but the poor performance of the NN was surprising. As was discussed in the results section of the NN, we hypothesize that there may be several potential reasons for this.

There were a number of challenges in the project. Specifically, in the absence of GPU's the team faced great difficulty during feature processing, cross-validation, and training stages due to the computationally expensive nature of working with so many images. Another challenge came in the form of debugging our neural network based learners. Because of the complex nature of NN's and CNN's we found it difficult to track down bugs in our code, especially in the case of the NN implementation.

One way where we improved the prediction results is by including more images into the training dataset. The training dataset was imbalanced with 8000 images for the Class 0 and the classes 30 to 39 had only 200 images. Due to this, there were a lot of false positives which were reported by the CNN model. During our successive iterations, we included few more images to the existing training dataset and we were able to observe that our prediction accuracy rose from 83% to 89%.

Increasing the number of training steps wasn't a good option as we lacked the infrastructure to support the process. Another technique would be to include distortions (deforming, cropping or brightening) in the images thereby expanding the training dataset. However, the training process took a lot of time as the bottleneck caching is no longer useful and the input images are not reused. Hence, we decided to drop this idea.

Other hyper parameters which we used in Convolution Neural Networks are the learning rate and the training batch size. The learning rate is used to control the magnitude of updates to our last layer during the training process. That being said, lower the learning rate, longer is the learning process, better is the precision. The train batch size refers to the number of images that must be examined during each training step.

Future, explorations would include exploring more

hyper-parameter settings for our CNN to improve its performance. In addition, we would like to further explore the hyper-parameters of our NN implementation and create a more efficient coding of it. Other explorations might involve exploring different network architectures by using a pre-trained deep neural network and pairing it with other learning algorithms like SVM's.

6. Statement of Contributions

The work was equally distributed amongst all the team members. All participated equally in deciding the preprocessing of data and features' selection. Senjuti performed the baseline learner, consisting of Logistic Regression, and performed feature extraction using SURF. David Newton implemented the neural network. Nirmal implemented the Transfer learning using Convolutional Neural Networks.

References

[1] C. Harris, and M. Stephens. *A combined corner and edge detector.*, Alvey vision conference. Vol. 15. No. 50. 1988.

[2] J. Canny. *A computational approach to edge detection.* IEEE Transactions on pattern analysis and machine intelligence 6 (1986): 679-698.

[3] A. Krizhevsky, I. Sutskever, and G.E.Hinton. *Imagenet classification with deep convolutional neural networks.* Advances in neural information processing systems. 2012.

[4] Tensorflow: *How to train Inception's final layer for new categories*

[5] Google Code Labs: *Tensorflow for Poets*

[6] Ariadna Quattoni, Michael Collins, and Trevor Darrell *Transfer Learning for Image Classification with Sparse Prototype Representations* IEEE Conference on Computer Vision and Pattern Recognition, 2008.

[7] Our Repo: *Link in README.md file*