COMP551 Kaggle
Classifying hand-drawn images

Daddy Science
Kristy Mualim 260679030 kristy.maulim@mail.mcgill.ca
Kabilan Sriranjan 260671847 kabilan.sriranjan@mail.mcgill.ca
Richard Zhang 260664443 richard.zhang@mail.mcgill.ca

I. Introduction

The goal of the Kaggle Competition was to classify images from Google's *Quick, Draw!* dataset. The dataset contained hand-drawn images of things pertaining to 31 different categories. A baseline linear SVM classifier, feed forward neural network trained with backpropagation, and a few different implementations of convolutional neural networks were all used and evaluated on the dataset.

Provided with over 10,000 labeled training samples and 10,000 test samples, it was observed that the CNNs we implemented performed significantly better on this image classification task relative to both the Feed Forward Neural Network and Baseline Linear SVM Classifier.

II. Feature design

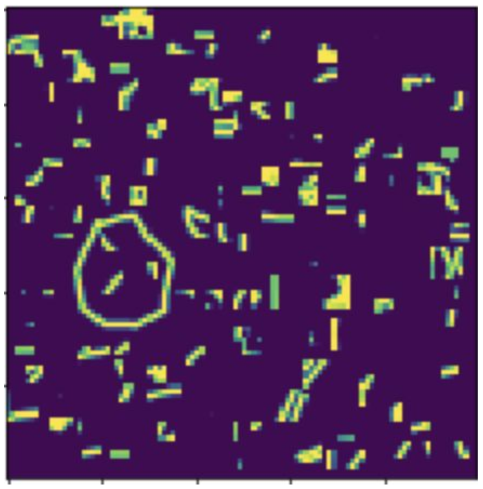


Fig 1. Raw input images from *Quick, Draw!* dataset

A. Noise Reduction

Each image in the dataset had random noise distributed across it. Given this additional noise, we seeked to apply noise reduction by applying a threshold on pixel intensity. Pixel values above 150 were increased to 255. The remaining pixel values were set to 0, reducing background noise. After which, we applied a means to locate all the connected components located in Fig 2 below.

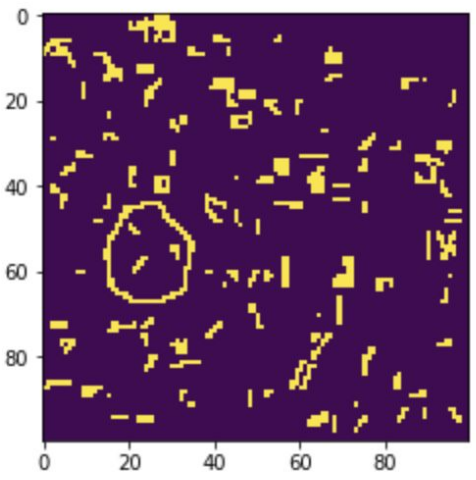
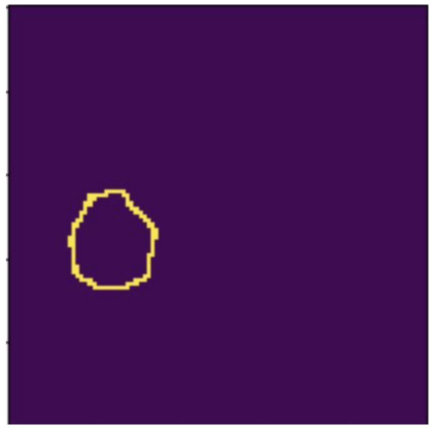


Fig 2. Input image after noise reduction

B. Image Extraction

The image is isolated by extracting the respective width and height values of all the connected components found in Fig 2. The connected component of interest was then selected by finding the connected components that registered the largest area (width * height). Utilizing a blank canvas, we then copied over the pixels that corresponded to the connected component of interest to obtain Fig 3, where the image is completely free of noise.



1

Fig 3. Image processed for noise reduction and extraction via finding largest boundary box

This image then underwent further preprocessing, where the coordinates of the box governing the connected component of interest was used to reshape the image into a 34x34 pixel image as illustrated below, Fig 4. We chose 34x34 as that is a standard size for CNN and image recognition.

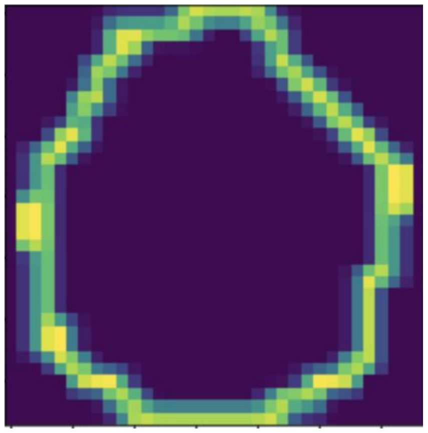


Fig 4. The picture's dimensions are then transformed to contain a 34 x 34 boundary of the image.

C. Data Augmentation:

We utilized a data augmentation feature as a means to prevent our CNN from overfitting the training data. Data augmentation increases the amount of training data by performing slight transformations like reflecting, rotating, and shifting and then adding the transformed data to the original dataset. Since any image's class stays the same under these minor transformations we have essentially introduced new labelled training samples.

We used ImageGenerator from Keras to rotate and flip our images. We believed that having images of different orientations will allow for the CNN to better recognize higher level features. We allowed for both horizontal and vertical flipping, as well as a small degree of rotation of 15 degrees.

D. Principal Component Analysis

Principal Component Analysis (PCA) is a technique used to project data on a lower dimensional space while preserving the maximum amount of information. By taking each 34x34 image and converting it into a 1156 dimensional vector we can use PCA to represent almost the exact same image but with a smaller vector. By representing our data in a more compact way we would have fewer parameters to learn and hence overfit less. The idea is to find a subspace to project the data onto using the training data, and then to make a prediction we would project the test sample onto the same subspace and use the model on the projected test sample.

It is possible to visualize exactly how much information is lost by doing this by performing an inverse transformation on the projected data and plotting the image. See Fig 5 to observe our visualizations.

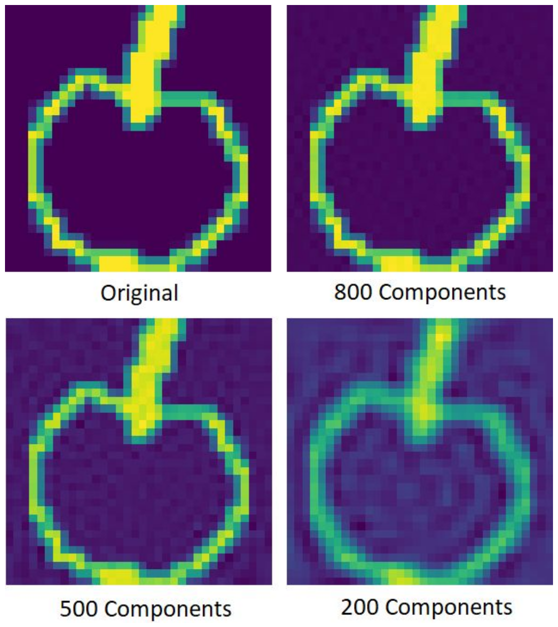


Fig 5. Visualization of information lost by projecting onto 200, 500, and 800 dimensional space from the original 1156 dimensional space

This projection only makes sense when the input is a single vector, not a 2D grid. If we were to reshape our compressed images into a matrix we would lose information about the which pixels were next to each other in the original image, something that CNNs need to use. Therefore it would only make sense to apply this technique to models like the Support Vector Machine or Logistic Regression. When we observed the poor performance of the SVM and the LR we suspected it was due to underfitting, which PCA does not help with. Hence we chose not to go further with this idea.

III. Algorithms

The central algorithms used in this Kaggle classification problem encompasses a baseline linear SVM, a feedforward neural network trained on backpropagation and various implementations of CNNs.

A. Linear SVM

Linear Support Vector Machines (SVMs) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Classification is achieved by realizing a linear separation surface in the input space. The decision plane is created so that it separates labels by the largest margin.

B. Feed Forward Neural Networks

Feed Forward Neural Networks are created by utilizing multi-layered networks of sigmoid units. This means each layer is a collection of non-linear activation functions, so the overall network can model non-linear functions. The overall architecture consists of an input layer, an output layer and k hidden layers, such that the neurons present at the i^{th} layer form the inputs to the neurons at $(i+1)^{th}$ layer. The output layer then gives the predicted outputs.

We utilized backpropagation as a means for weight updates, where the error obtained upon our output prediction values are fed back through the network to allow the algorithm to adjust weights of each connection in order to reduce the value of the error function by a small amount.

This is performed multiple times either until the network converges or the maximum epoch number is reached.

C. Convolutional Network Networks

Convolutional Neural Networks (CNNs) form a class of deep, feed-forward artificial neural networks, usually applied to analyzing visual imagery. CNNs use a variation of multilayer perceptrons designed to require minimal preprocessing, indicating that the network learns the filters rather than having them be hand-engineered.

CNNs, similar to feed-forward neural nets, consist of an input and an output layer, with multiple hidden layers. Each individual hidden layer can be one of a convolutional, pooling, fully connected or normalization layers.

Convolutional layers apply a convolution operation to the input and passes the output as input to the next layer. Each convolutional neuron processes data only for its receptive field. The convolution operation applied helps to reduce the number of free parameters, allowing the network to be deeper in the second layer. After each convolutional layer, it is conventional to apply a nonlinear layer in an attempt to introduce nonlinearity to a system that was previously computing linear operations.

Pooling layers may be either local or global, both of which help to combine the outputs of neuron clusters at one layer into a single neuron in the next layer.

Fully connected layers connect every neuron in one layer to every neuron in another layer, its main principle is similar to a traditional multi-layer perceptron neural network.

CNNs are usually deemed more efficient than feed-forward neural networks in relation to image classification given its use of local receptive fields, parameter sharing and pooling. Local receptive fields help to focus on important features isolated to a small region of interest in the image. Parameter sharing is used by CNNs to control the number of parameters and is performed under the assumption that a feature useful to compute at a spatial position (x,y) would also be useful at a different position (x_2,y_2). Thus, neurons in each depth slice are fundamentally constrained to using the same weights and bias.

Training for CNNs is similar to that of feed-forward neural nets and involve backpropagation constituting a forward pass of training data, a loss function to evaluate error, a backward pass and a weight update.

IV. Methodology

A. Training, Validation Splits

The labelled dataset was split into 3 subsections to be used for model training, validation, and testing. 85% of the dataset was used for training, 12% was used for validation, and 3% was used for testing. While our test set is small, this is mollified by the fact that the final test set will be the unlabelled images. Our test set was primarily used as a verification measure on the scores of our validation set.

For our hand coded NN, we employed 10-fold cross validation to determine optimal hyperparameters such as number of layers and nodes in each layer. In 10-fold cross validation, the dataset is broken into 10 sections, with each section being used as the test set once for a total of 10 runs total. The average performance measured by F1 is taken over the 10 runs.

B. SVM hyperparameters

We tested a range of values between 1 and 10 for the penalty parameter C. The best value was determined to be 1, with a validation F1 of 0.27 on our preprocessed dataset. We decided to use L2 regularization, as the data was already sparse, and each pixel may have some impact on the output result. As we used LinearSVC from sklearn, we set dual=False to solve for the primal problem because our number of samples (8500) eclipsed the number of features (34x34).

C. NN Architecture and Hyperparameters

We tested a range of values for both the number of hidden layers, the number of nodes for each layer, and the learning rate. We tried between 1 and 3 hidden layers, with the number of nodes being between 1001 and 101, by increments of 200. Also, we tested alphas between 0.01 and 1.

We tested epochs of 1, 10, and 100. We used a batch size of 1.

D. Hand Coded CNN

Alongside using pre-trained CNNs found online we also tried building our own. The architecture and hyperparameters were all chosen by us through evaluating performance on the validation set. The initial architecture we had was inspired from an online Keras tutorial. It consisted only of two 2-D convolutional layers separated by max pooling layers, a fully connected layer, and an output layer. The number of nodes in the fully connected layer, the number of channels, the kernel size, and the stride size all were moderately tuned using the validation set. From there we implemented dropout layers and then further tuned hyperparameters.

Dropout is an approach to regularization in neural networks to help reduce interdependent learning amongst neurons and is implemented on hidden units during the training phase of sets of neurons chosen at random. The addition of dropout in our CNN was another means to prevent overfitting. The fully connected layers that occupy most parameters have a tendency for their neurons to develop dependency amongst each other during training, limiting the individual ability of each neuron resulting in overfitting. It should be observed that the addition of dropout allows a neural network to learn more robust features that are more essential in conjunction with other subsets of neurons.

For our selection of hyperparameters, we tested number of layers, the ordering and number of convolutional and pooling layers, batch size, number of epochs, and the addition of dropout layers.

ReLu was chosen as the activation layer of choice given its computational efficiency and helps to alleviate the vanishing gradient problem; the fundamental issue when lower layers of the network train very slowly due to the gradient decreasing exponentially through layers.

E. Pre-trained CNN models

There is a great deal of choice when it comes to the architecture of the CNN. Not only is there the consideration of the number of layers, but there is also a choice between of the type of each layer. Luckily, image recognition is a highly studied field and so there are models available for use in the Keras library with pre-trained weights. This is beneficial in terms of deciding CNN architecture, but there is also evidence that using pre-trained weights increases the accuracy of the network faster compared to training from scratch.

We tested InceptionV3 and DenseNet with pre-trained weights. Both of these models required 3 channels, while our pictures are black and white. Thus, we needed to convert the black and white images to RGB. Also, InceptionV3 had a minimum picture size of 75x75 pixels, but our images were 34x34 after preprocessing, so we scaled them up.

There is also a choice to the number of epochs to train. We found that a large epoch caused the models to grossly overfit on the training data. We were able to conclude this because the pre-trained models also show validation loss. After 15 epochs, the validation error cost begins to increase, while the training error continues to decrease - a clear sign of overfitting.

V. Results

A. SVM results

We achieved a range of F1 validation values around 0.05 on unprocessed data. After using preprocessed data, our F1 validation values clustered around 0.26. Specifically, for a range of c values between 1 and 10, the F1 validation values ranged between 0.27 and 0.26, with the best c value being 1.

B. Hand NN results

We found with more hidden layers, more nodes between each layer, and more epochs it took far too long for the NN to run. As such, we do not have complete information on the validation performance of the NN. Our neural net with 2 hidden layers of size 101 nodes, on 10 epochs and a learning rate of .05 had an accuracy of 0.32.

C. CNN results

C.I. Optimizers

We utilized an architecture that involved incorporating two 2-D convolutional layers, between max pooling layers and added a single dropout layer before the final output activation layer to achieve an accuracy of 0.70 and registered a final loss of 1.6788. This model was trained with the Adadelta optimizer.

Alternatively, using the same architecture as mentioned above, we utilized a different optimizer (Nadam) to evaluate how the CNN would differ in performance. It was observed that utilizing Nadam with default parameters as an optimizer achieved a slightly lower accuracy score of 0.663 and a loss of 1.49.

From these results we decided to only do further analysis with the Adadelta optimizer.

C.II. Dropout layers

Regarding the inclusion of more dropout layers, we applied a dropout layer after each convolutional and max pooling layer. This decreased validation set accuracy to 0.623, indicating an excessive data loss. Hence our final model only had one dropout layer.

C.III. Epochs

We tested different epochs on our most basic CNN, which had no dropout layers, and only 3 consecutive convolutional, convolution, pooling layers at the batch size of 100. The best validation accuracy occurred at around 15 epochs.

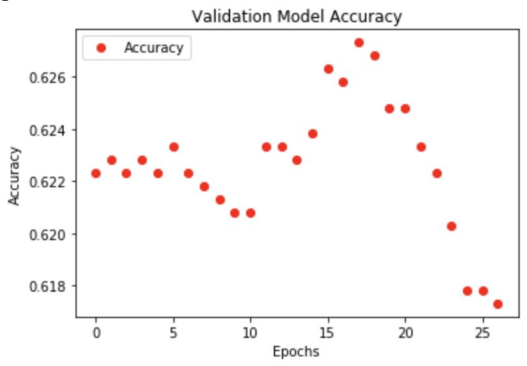


Fig 6. Number of Epoch against Validation Accuracy Batch size

Before adding image rotation and flipping, we found that overfitting began at around 15 epochs for both pre-trained and self-composed CNN models. However, after adding the rotations and flipping, the larger dataset allowed for greater epochs before overfitting emerged. The following graph shows the validation loss on 35 epochs with the augmented dataset.

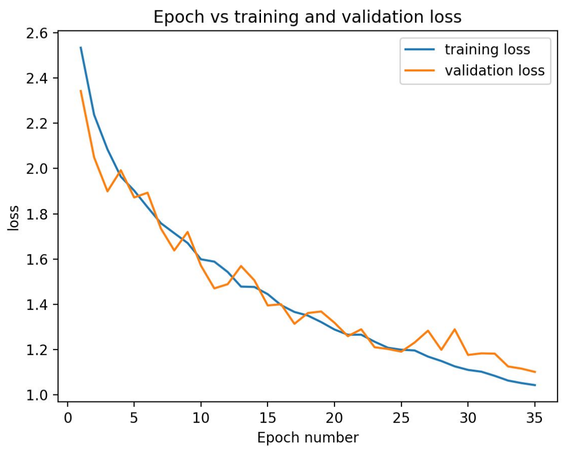


Fig 7. Loss per epoch

In addition to using the larger dataset with rotations and flipping, the modification of the most general CNN architecture, with the addition of convolutional layers and a dropout layer found that both accuracy and loss registered relatively small changes after 5 epochs. The following graph shows the validation loss and accuracy on 20 epochs with the augmented dataset, illustrating the loss is steadily increasing with increasing epochs.

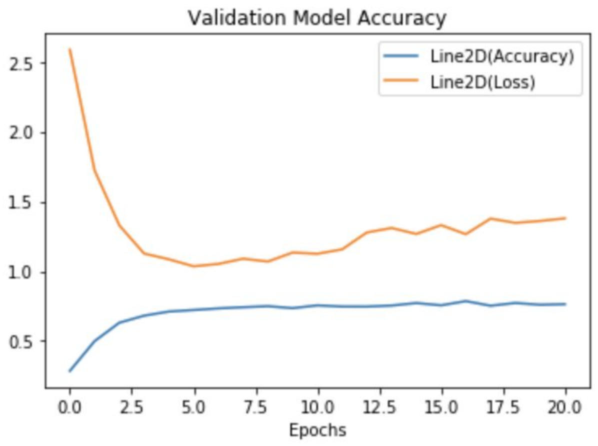
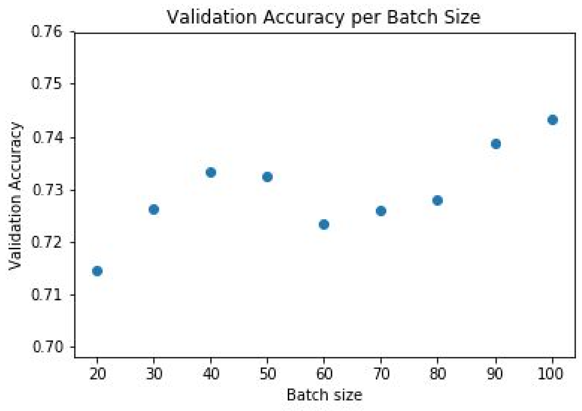


Fig 8. Validation Metrics against Number of Epochs

C.IV. Batch sizes



After doing the analysis to find the optimal number of epochs we fixed epoch number and tried to find the optimal batch size for the learning algorithm.

Our final hand-tuned CNN with flipped data and tuned batch size and epoch number had a testing accuracy of 0.721.

C.V Pre-trained Models

With the pre-trained CNN, there is an option to only freeze certain layers. When only unfreezing the top layer, we had a validation accuracy of around 0.03 for both DenseNet and InceptionV3. This is unsurprising, as these models had imageNet weights, which were coloured images, and not hand-drawn. Increasing the number of layers to retrain increased validation accuracy, until 0.65 and 0.6 for InceptionV3 and DenseNet respectively when retraining all layers.

After specifying the evaluation metric to be categorical accuracy, we reached 0.67 on the test set (submission) for InceptionV3.

Using flipped and rotated images along with original dataset, we achieved 0.684 F1 value on the validation set.

After adding a dropout layer, F1 value increased to 0.701. Finally, increasing the number of epochs lead to a final testing accuracy of 0.724.

C.VI. Comparison of Methods

Both hand tuned and pre-trained CNNs had similar performances at around 0.72. These CNNs performed much better than NN and SVM. NN and SVM performances only had about 0.05 difference in F1 scores, which may be because we were not able to get many results from the NN with larger epochs, more layers, or more node counts due to time issues. Thus the full capabilities of NN may not be expressed with our results.

VI. Discussion

A. Proprocessing

With our knowledge of preprocessing hand-drawn datasets, we realized that for one image in particular (eg. rabbit), the image itself was cut off due to the boundary threshold being set too low. Boundary thresholds served as a means to remove noisy pixels. We assumed that the image of interest would have pixel intensities above a certain threshold. There was a trade-off in determining threshold value and obtaining full images for all the training datasets. Thus, there could be an alternative way to both remove all noise while maintaining the integrity of all the respective images. Image preprocessing improved performance for all classifiers.

B. *SVM*

SVM was unable to capture the complexity and nonlinearity intricacies of the dataset and thus performed with a low accuracy of 0.26. The addition of non-linear basis functions to enhance performance could be a plausible option. However, as evaluated by our results, it seems that there are other more state-of-the-art models that could be implemented that would achieve better accuracies even without prior hyper-parameter tuning.

C. *NN*

The biggest limitation to our hand-written NN was the speed. The slow speed made rigorous testing of hyperparameters infeasible. Future works could involve improving the efficiency by using parallel processing, or by using GPU.

D. *CNN*

We wanted to leverage existing models for our CNN. We tried InceptionV3, and DenseNet models. While using these models reduced the search through NN architecture, there were still some difficulties. The biggest was reconciling input sizes and formats. Both InceptionV3 and DenseNet required coloured images, while our images were black and white. In addition, InceptionV3 required that images be greater than 75x75 pixels. However, after preprocessing, our images were only 34x34 pixels. Scaling the images had its own downsides, mostly being that the images became more blurred and less recognizable after being scaled up.

An additional issue with the pretrained weights is that they were trained on imageNet data, which were coloured images, and not directly transferable to the hand-drawn dataset for this competition.

The greatest advantage that we achieved from applying our hand done CNN was the ability to fine-tune our hyperparameters. The initial implementation of a general CNN achieved a test accuracy of 0.602. Further improvements, involving tuning hyper parameters as mentioned in methodology, to the CNN achieved a validation accuracy of 0.771.

E. *Possible Limitations*

The runtime of the different architectures we tried took a long time on CPU, drastically limiting our means to optimize our models.

We could have found better combinations of hyperparameters could had we done a more exhaustive grid-style search rather than optimizing for each hyperparameter sequentially.

F. *Future Work*

Implementing our models on a GPU would vastly improve the runtime, allowing us to better hypertune parameters and adjust our models to enhance accuracy. For example we would be able to more feasibly do a grid search on hyper parameters as mentioned previously.

The parameters we tried and tested in our hand done CNN model utilized just a couple of ways to select parameters for a CNN model. Apart from the Max Pooling we performed, other options for pooling layers include average pooling and l2-norm pooling.

We reached a trade-off in determining threshold value and obtaining full images for all images of training data. We found at least one image that got truncated due to the threshold being too high. Thus, future work could be to investigate alternative means that would allow more effective preprocessing, in which both noise is reduced and image integrity is maintained.

An alternative implementation to the proposed CNN would be Bayesian Inference. In situations where limited data is available, the data alone might not be sufficient to specify a unique solution to the problem. Hence, priors introduced with Bayesian method could help guide towards a preferred solution. The bayesian approach seeks to incorporate prior knowledge in data analysis and revolves around posterior probabilities to summarize the degree of one's certainty concerning a given situation. Its application in image classification has been rapidly gaining exposure and usage, and thus could be an effective means in relation to the given dataset.

VII. Contributions

The authors, Kabilan Sriranjan, Kristy Mualim and Richard Zhang all took active roles in contribution to this project.

Kabilan Sriranjan implemented CNN from scratch, attempted PCA, data augmentation and contributed to the composition of this report.

Kristy Mualim did image preprocessing and data augmentation, implementation of the CNN from scratch and fine-tuning, the baseline classifiers, and contributed to the report.

Richard Zhang implemented the regular NN from scratch, tuned the pre-trained CNNs, helped with the baselines, and contributed to the report.