Magic the Gathering Card Image Recognition with Deep Learning
Jeff Patterson, Dustin Lehmkuhl

Overview / Abstract:

Computer vision is a quickly growing field using various elements of deep learning to classify images. We want to use some of these techniques to classify Magic the Gathering cards. Magic cards have many features making them ideal samples for image processing and deep learning classification methods.

There are a finite number of magic cards printed, each with its own particularly unique artwork. Therefore, there is a known number of mutually exclusive outputs to the network. This makes the process of training and testing the network fairly straightforward. The format of a magic card is also very regular, with all important card-by-card information appearing in the same place. This does not change for any traditional magic cards made in the modern era. Since the artwork regularly appears in the same place every time, it simplifies the extraction of the most meaningful features in a card for image processing and deep learning applications.

For magic card recognition software to be useful to vendors and magic collectors, the input image must be allowed to take any size (x,x,3) representation of the card art as an input. A user of this program will not take pictures with the same camera, or at the same distance away from a card as another user. The user must be able to snap a picture of the card, and receive an accurate prediction of the card name.

One way to keep the size of the input of a neural network constant is to preprocess the image before insertion into the neural network. Some common techniques involve scaling the image to the required input dimensions or generating statistical data from the image and training the neural net to classify based on those features. For this project, we decided to use a process called image hashing.

An image hashing algorithm takes an image and sequentially runs statistical tests to derive features from the image. These algorithms have two very important features for deep learning:

1: Images that have slightly different hash values will only vary slightly. The more different the hash value, the more different the images are with regards to the statistical measurement being made.

2: Image hashing algorithms function independently of initial image dimensions.

These two properties are critical for their use in neural network development. Since the “Likeness” of two images can be determined by the similarity of their hashes, a loss algorithm can be designed to input two image hashes and output a representation of their similarity. For our application, the fact that these functions act the same despite picture dimensions means that any picture taken by the person using the software will be a valid input to the neural net.

Application:

For this project, we are computing the image hashes of hundreds of procedurally generated magic card artworks with various modifications made to them to simulate wear, printing differences, and camera issues. These are used to train and test the neural network. Since the input to the network is only a single line of ~130 doubles, these networks do not take very long to train as compared to other neural network architectures.

Sample Acquisition:

The generation of the training data began with the collection of every card in our test test, a total of 264 cards. The magic card images are open source and of a standardized format; all that was required was a download. The program used to download the images was Gatherer Extractor 4 - a program specifically designed to generate custom Magic: The Gathering databases. This program extracted a .png file for every card in the test set labeled with the title of the card. Neural Networks can be significantly limited by the quality of their training data, and by using this approach of auto-labeling the integrity of the input data could be maintained.

Image hashing checks the similarity between two images, therefore it is crucial to sample the card in such a way that there is maximum difference between each sample. Take these two cards for instance:

There are many similarities between these two cards. The layout of the card is the exact same, and the background colors of everything except the card art is identical. When the image hashing algorithm checks the similarities between the two cards, it's going to find many. This could result in unnecessary misclassifications, which we're trying to minimize.

To reduce the number of similarities between every sample, we extracted the most unique aspect of every card and instead used that image for training.

Despite how similar the previous images appeared to anyone who isn't familiar with magic cards, once the unique card art was extracted the difference between the two is abundantly clear.

The results of a single image hash are a line of 16 hexadecimal features. Using 16 features to classify 264 unique cards would result in serious underfitting issues. A method had to be developed to generate additional meaningful features.

Two simultaneous approaches were taken:

1: Use multiple image hashing algorithms to extract more data from each image.

Image hashing algorithms are very unique, and are programed to characterize different aspects of a picture. By using many image hashing algorithms independently and appending them together into a single long hexadecimal word, a larger number of meaningful features can be extracted.

2: Split the image into numerous sub-images based on color, and generate image hashes for each color's sub-image alongside the full image hash.

The input to the image hashing algorithms used is an (X,X,3) image architecture, meaning that 16 features are generated from an (X,X,RGB) image characterizing the input. If 3 additional images are generated by converting the (X,X,RGB) image into (X,X,RRR), (X,X,GGG), and (X,X,BBB) subimages, each color's intensity plot can be image hashed to identify further unique similarities.

For example,

Pictured above is Aerial Responder's 4 input images. Upper left is the full card art, upper right is the red intensity map, lower left is the green intensity map, and lower right is the blue intensity map. All 4 of these images are run through two image hashing algorithms, P-Hashing and D-Hashing. After the hashing is complete, and all of the results are appended together, a 128 feature hexadecimal word is created for training.

Perceptual Hashing (P-Hashing) is a classification of image hashing algorithms that rely on performing a Discrete Cosine Transformation on a shrunken version of the image to characterize the presence of certain frequencies, The result of this image is a square representation of how much of each spatial frequency is in the image. Then the average of all of the "pixels" is computed, and each pixel is measured to be either above or below the average. Pixels above the average are given the binary value 1, and all others are given the binary value 0. These are then grouped and converted into hexadecimal values that, when appended together, create the image hash.

Difference Hashing (D-Hashing) is another classification of image hashing algorithms that searches for differences in regional pixel intensity. Once again, the picture is reduced in size. Then, the difference between each pixel is calculated. Much like p-hashing, these pixels are assigned a value based on how they compare to the average for the picture, and are grouped sequentially to generate hexadecimal values that make up the image hash.

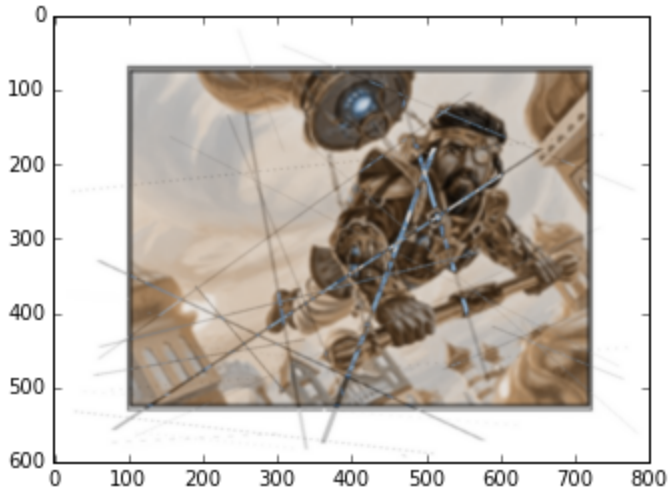
These two image hashes were selected as the initial testing algorithms out of convenience due to their ease of implementation with the python extension ImageHash. There were two other possible algorithms to use (average hashing and wavelet hashing) but initial tests with these image hashes showed much less variation between each card.

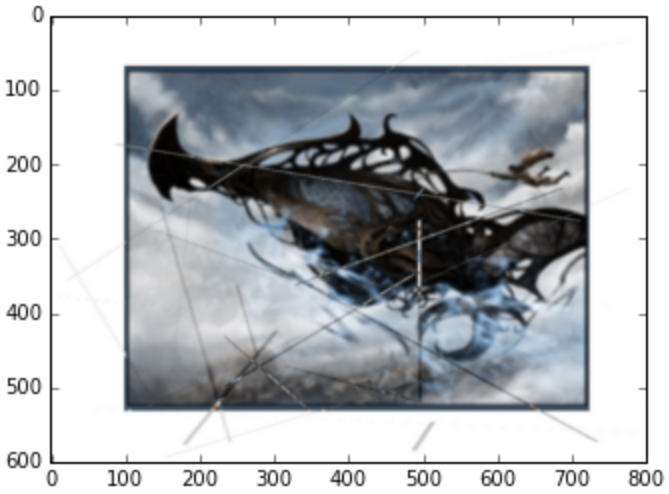
Procedurally Generated Noise:

We generated 79200 images (264 card names, and 300 images per card name) by using the following algorithms and functions:

1) Addline() - Generates a line across random space of the card and reducing the RGB values of that space to indicate a scratch of some kind. For each image this is performed from 0 to 100 times based on a uniform distribution.
2) scipy.ndimage.filters.gaussian_filter() - applies a gaussian filter with a specified standard deviation to the image creating a blurring effect. This standard deviation was set as 0 to 1.5 on a uniform distribution

Of course more noise-generating functions can be added, for example we started with rotation and determined that hardware would remove this concern. This data creation took roughly 4 hours to complete. Below are some images affected by this noise generation:





Application of Image Hashing / Preprocessing:

Typically hashes are described in terms of a 16 hexadecimal character long word. For each noise-applied image, we converted the R, G, B, and full images into both *d-hashes* and *p-hashes*. Since tensorflow does not recognize these hashes as anything but strings, each hash was separated into 16 pieces, converted into an integer (from 1 to 16), and applied as column features. This leaves us with (4 color schemes)*(2 hash types)*(16 characters) = 128 features which can be scaled and matched with card name labels. The labels were preprocessed by one-hot encoding the 264 card name labels into a matrix (200 images)*(264 labels) for the training set and (100 images)*(264 labels) for the validation set.

Neural Network Design and Results:

Using one hidden layer neural network of 256 nodes and a relu activation layer:

```
Epochs =300, Training Set Size =52800, Nodes =   256, Alpha = 0.0000, Batch Size =
52800, STD = 0.100
```

		cross-entropy		error-rate			
	epoch	training	validation	training	validation	L2	time (min)
300	0	5.85669	5.85707	0.997	0.998	0.000	0.0
300	30	1.45093	1.45758	0.081	0.085	0.000	0.2
300	60	0.09357	0.09711	0.003	0.004	0.000	0.3
300	90	0.03798	0.04135	0.001	0.002	0.000	0.4
300	120	0.02485	0.02827	0.001	0.002	0.000	0.6
300	150	0.01810	0.02155	0.000	0.001	0.000	0.7
300	180	0.01389	0.01733	0.000	0.001	0.000	0.9
300	210	0.01104	0.01445	0.000	0.001	0.000	1.0
300	240	0.00901	0.01238	0.000	0.001	0.000	1.2
300	270	0.00751	0.01083	0.000	0.001	0.000	1.3
300	300	0.00636	0.00963	0.000	0.001	0.000	1.5

Where no stochastic gradient descent or regularization was used. With minimal computation time, we achieved a near-perfect accuracy (>99.9%) of our validation set. This indicates that this technique can handle much more noise and many more cards in the future. This also eases our concerns that utilizing hash information significantly reduces information about the image leading to lower prediction accuracy.

Conclusions / Further Study:

Overall, we are quite impressed with the performance of a simple neural network design to classify our 256 cards. A natural extension of this process would be to incorporate every card from Magic's history, as well as attempting to read minor differences between editions and alterations. One interesting bonus of this particular application is that there is 100% label accuracy, which means that every label is what would be applied if a human were to classify an image. This allows for a computer to theoretically achieve a perfect prediction accuracy as it is not limited by the data being incorrect.

We anticipate progressing with this project by developing more rigorous (and hopefully faster) algorithms for noise generation, allowing for minor extra pieces of a card's anatomy (like card name, mana cost, set symbol, etc.), and development into a mobile app. The most limiting process is easily the data generation, as the software we use to convert image hashing requires a saved jpg file on the local disk, which we then have to delete. That step alone takes 80% of the time it takes to generate each dirty image and convert their hash information. The computational requirements for developing a prediction of an image with an already trained neural network are minimal, making this project an ideal application for a smart phone.