Computer Vision for Card Games

Matias Castillo
Department of Electrical Engineering
Stanford University
Stanford, California 94305
matiasct@stanford.edu

Benjamin Goeing
Department of Electrical Engineering
Stanford University
Stanford, California 94305
bgoeing@stanford.edu

Jesper Westell
Department of Electrical Engineering
Stanford University
Stanford, California 94305
jesperw@stanford.edu

***Abstract*—For this project, we designed a computer vision program that can recognize and accurately classify photos of playing cards lying on a table into one of 52 classes. In conjunction with our CS221 project, where we designed a card playing AI, this allows us to develop a holistic application for a player with a real-world card deck to play against a computer or for a player to use AI support (e.g. via Google glasses), playing a real world game against an adversary. We decided to test a variety of machine learning methods to tackle this computer vision problem. Using classical methods such as multiclass logistic regression and multiclass support vector machines with regularization, we managed to achieve accuracies between 82-89% on the dev and test sets. Using deep learning approaches, we designed a custom convolutional neural network in Tensorflow, that manages to get 100 % on the dev and test sets.**

Keywords*—*Computer Vision, Object Classification, Convolutional Neural Networks

I. Introduction

Most programs that use artificial intelligence for card games are focused on intelligent agents and smart decision making in a virtual environment. Not many study the interaction of the agents with the real world. In this project we used different machine learning models for recognizing individual playing card images taken by a smartphone. This could be useful, for example, for automatically analyzing a high stakes card game on TV, based on the visual input. Another application would be to combine this program with Google glasses to find the best move in a real-world situation, such as a game of poker or blackjack to turn the odds in your favor.

II. Data Augmentation and Engineering

Most of the data sets available on the internet only contain perfectly aligned images of playing cards. For our application, we were interested in being able to recognize real images taken with a smartphone's camera, to simulate a real-world application. For this purpose we decided to develop our own data set.

We purchased a deck of cards and took our own pictures. For the data set generation, we first took 10 images of each card class considering different positions and distances, for a total of 520 images. Then, we developed an image augmentation algorithm that transformed the 10 images of each card to a set of 1.050 images of the same card, by using different combinations of rotation, zoom, brightness, contrast, sharpness and color manipulations. We ended up with a 1.6 GB data set of 54,600 card images. A small sample of the data set can be seen in figure 1.

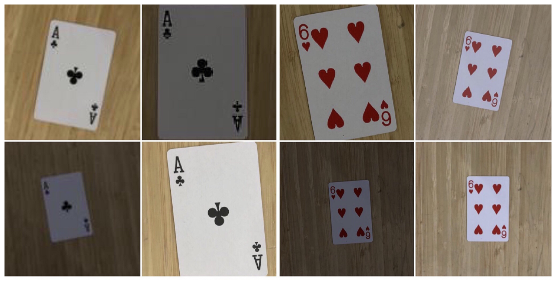


Fig. 1: Example of the data set

Each of the images was originally represented as a tridimensional (480, 640, 3) array of RGB pixels. We decided to re-scale it to (160, 120, 3) for better fit in memory during train time and to also normalize by dividing by 255. Additionally, we defined a Y-vector with the correct labels corresponding to one of the 52 possible classes (4 ranks and 13 suits).

We decided to divide our data set into 90 % train (49,140 images), 5% dev, and 5% test. For this purpose, we generated random indices and wrote a custom script to read from the correct images and labels. It turned out that loading all training examples into memory at the same time was not possible, so we developed another script, loading only one randomized minibatch at a time into memory. This solved our memory problem, but significantly increased the time it would take to train models, due to the large number of memory access operations.

III. Related work

While there is a large variety of computer vision algorithms and applications, we have found nothing that deals with recognizing playing card images in particular. During the 2016 instance of CS 229, a project group developed a convolutional neural network to recognize and classify food images, which is similar to our project in nature [1]. However, different kinds of foods usually share very different features (e.g. colors, shapes etc.), whereas playing cards look more similar to each other. On the other hand, objects that belong to the same food class could vary in shape and size, whereas playing cards always have a rectangular element, and similar proportions, although they may be rotated. Both problems therefore present different challenges for a CV algorithm.

IV. Classical Machine Learning Models

As a first step, we decided to test a variety of classical machine learning methods, to gauge the complexity of the problem. We also wanted to establish a baseline of traditional methods to compare against deep learning methods, which we implemented as a second part of the project. We used scikit-learn to implement these methods.

A. *Models*

Multiclass Logistic Regression (MLR)

As a first method, we decided to use a version of logistic regression, that combines multiple binary classifiers in a "one versus all" (OVA) scheme. This means that the model has 52 binary classifiers that learn by minimizing the logistic loss. Then, when testing a specific example, the model chooses the best confidence score among all the classifiers.

The learning was done using stochastic gradient descent, and the loss function J can be expressed as:

$$J(w) = -[y^{(i)} \log(h(\phi(x^{(i)})) + (1 - y^{(i)}) \log(1 - h(\phi(x^{(i)}))]$$

where h is the sigmoid function, $\phi(x^{(i)})$ and $y^{(i)}$ are the feature vector and the correct classification for the example i, and w is the learned parameter.

Multiclass Support Vector Machine (MSVM)

This model was also implemented using multiple binary classifiers with OVA scheme and stochastic gradient descent, but now optimizing a hinge loss function in each of them. Therefore, the loss function J in each of the classifiers is:

$$J(w, b) = \max(0, 1 - y^{(i)}(w\phi(x^{(i)}) + b))$$

where w, b are the parameters of the hyperplane, and $\phi(x^{(i)})$ and $y^{(i)}$ are the feature vector and the correct classification for the example i.

B. *First results and adding regularization*

After some first experimentation, we realized that both of these models show signs of overfitting, as they achieved close to 100% train accuracy, but only 81.9% and 85.2% dev accuracy respectively (see Fig. 3 in the results section). To reduce overfitting, we decided to add regularization. We decided to use squared norm regularization (L2). For each of the classifiers, the regularized loss function J_r can be expressed as:

$$J_r = J + \lambda ||w||_2^2$$

In order to find the best values for λ, the models were trained using a broad range of λ. Figure 2 illustrates the achieved accuracy values on the dev set of each model after 30 iterations. The best λ obtained were 0.1 and 0.02 for the MLR and MSVM models respectively.

Finally, the performance of both regularized and non-regularized models was compared after training them for 30 iterations (see Fig. 3).

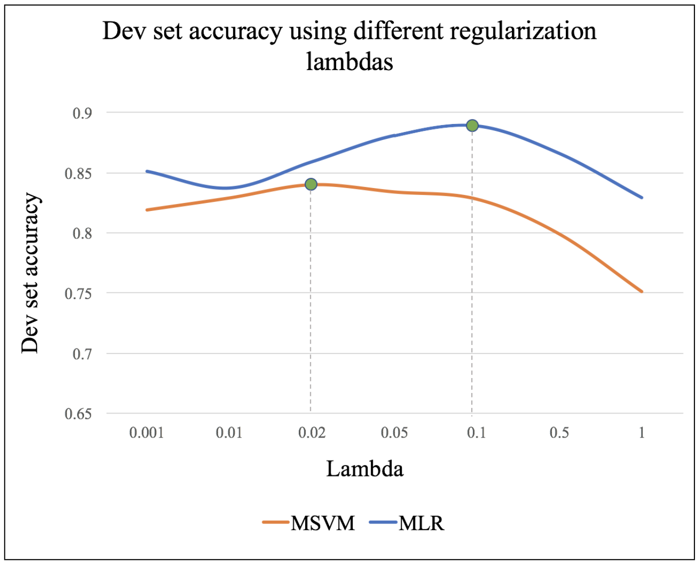


Fig. 2: Performance of MLR and MSVM models using different regularization parameter λ

C. *Results*

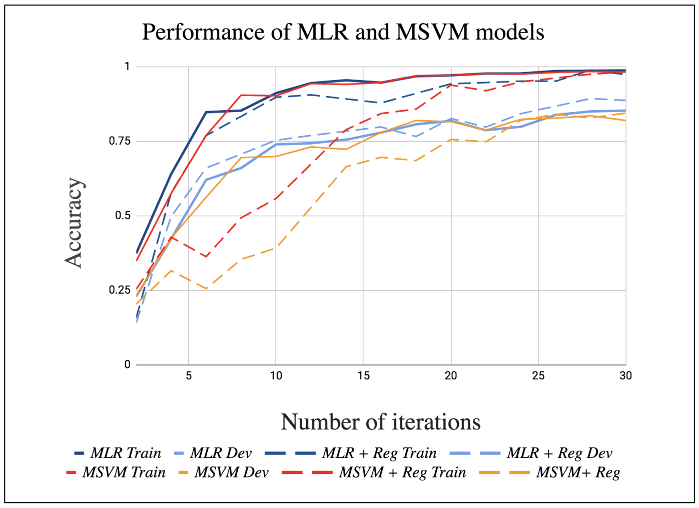


Fig. 3: Performance of classic machine learning models

The initial MLR model reached an accuracy of 98.7% on the train set and 85.2% on the dev set. The performance of the original MSVM model is slightly worse with 98.2% and 81.9% accuracy for the train and dev sets respectively. After applying L2 regularization, the models improved their accuracy by 3 to 5 percentage points, with a dev accuracy of 88.7% for the MLR model and 84.4% with MSVM.

V. Convolutional Neural Network

A. *Motivation*

Since we still didn't get close to achieving the desired 100% accuracy, we decided to go a step further, and attempt to design a convolutional neural network. Although we didn't have previous experience with the technique, aside from one team member concurrently taking CS 230, we thought that this would be the most promising approach.

B. Defining the architecture

While there are a number of pre-trained networks out there that we could have copied and pasted, we decided that we wanted to get the full experience out of this exercise. We therefore decided to design and train a network from scratch using TensorFlow. We decided on an architecture using two convolutional layers and one fully connected layer using the cross-entropy loss, as this architecture had worked well in a homework problem of CS 230 [2]. The cross entropy loss can be expressed as:

$$J = -\sum_{i=1}^{m} y_i \log \hat{y_i}$$

Figure 4 visualizes the high-level architecture of our network.

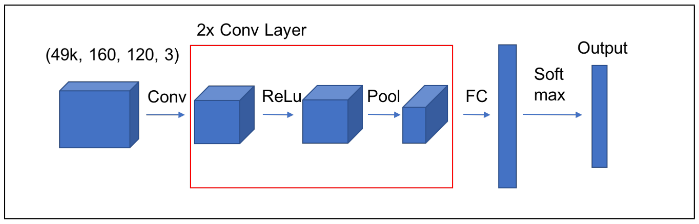


Fig. 4: Architecture of the Convolutional Neural Network

C. Defining and tuning hyper parameters

We experimented with a number of different values for the filters, poolings, and strides. For this purpose, we decided to run our model with only 5000 training examples, 1000 dev and test images for 5-10 epochs to compare performance of different hyperparameters (Running the full model would have taken to long for us to iterate quickly enough).

We found that changes of the hyperparameters for the convolutional layer didn't make too much of a difference. We maintained that the first convolutional layer should work with larger filters, to detect the edges of the cards, whereas the second convolutional layer should work with smaller values, to detect more details in the image.

We ultimately decided on the following details:

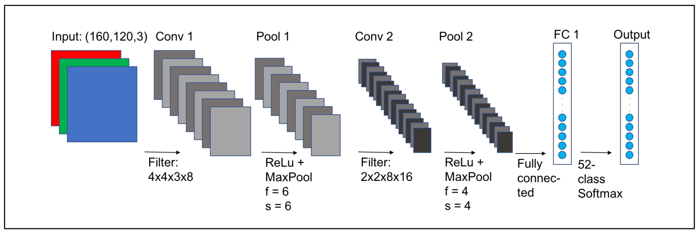
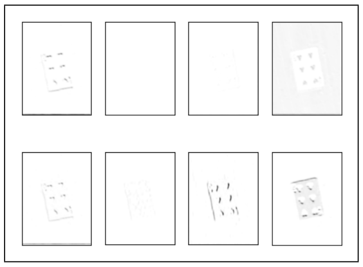


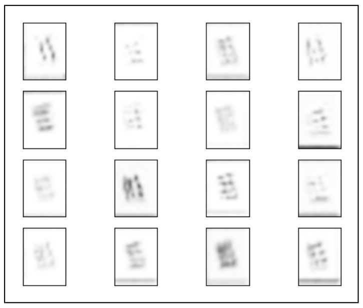
Fig. 5: Model details of the Convolutional Neural Network

The following figure visualizes the outputs of our 2 convolutional layers:

(a) Original image



(b) Saliency map Conv Layer 1



(c) Saliency map Conv Layer 2

Fig. 6: Propagation of a 6 of Hearts through the Convolutional Layers with 8 and 16 filters respectively

The hyperparameter that turned out to make a huge difference was the learning rate. Initially starting with a learning rate of 0.001 led to very poor results. We experimented with a variety of different learning rates, and ultimately decided to chose 0.01, as it yielded the fastest increase in accuracy with our "simplified" model with 5000 training examples. However, as this still took over 30 minutes to run, we did not sample a very large amount of learning rates, as this again would have taken a considerable amount of time. Instead, we decided to go with a learning rate of 0.01, as it initially seemed to be doing well, and decided to go back and refine it further in case the model would not perform as expected later on.

D. Results

With the architecture and hyper parameters from above, and a learning rate of 0.01 we ran the model over night for a total of 30 epochs, which took approximately 14 hours on our CPU, training on all 49,140 images. We ended up achieving 100.0% accuracy on the dev set after 17 epochs, and also 100.0% on the test set. While the algorithm converged to 99.6% on the dev set after only 7 epochs, it took another 10 epochs to get the remaining 0.4% right. Figure 7 shows our results:

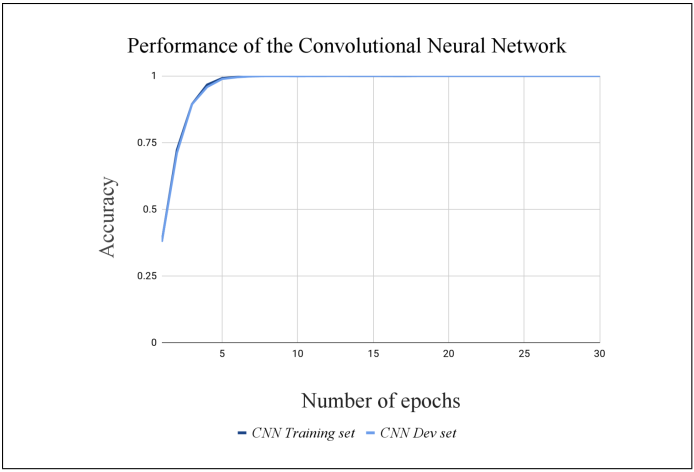


Fig. 7: Results of our convolutional neural network

VI. Discussion of results, conclusion and future work

We believe that the CNN worked better than the traditional methods because of the variety of zoom, rotation, angles etc. in the images, that the MSVM and MLR were not able to capture. We believe we could give the traditional methods another performance boost through additional pre-processing such e.g. axis aligning the images before handing them to the models. However, we believe that the CNN would also naturally perform better on this task, so our future efforts would likely focus on the CNN in practice.

We are very glad about our result of 100.0% with the CNN. However, we suspect that this could be because despite all of our augmentation attempts, the images still share a lot of similarities (e.g. all have a brown table as a background). To further validate our model, we scraped 10 card images of the internet and tested our model on these images from different distributions. We achieved an accuracy of 50% on these images, which shows that our model is still able to recognize these cards to some extent (given a $\sim$2% accuracy from random guessing), but is not performing as well on different distributions. Some sample images that were classified correctly and incorrectly can be viewed in figure 7. Note that many misclassified images were classified as having the suit of hearts. This could mean that the trained classifier has a bias towards this particular suit. The color red does stand out more than the color black, especially in the training examples with reduced brightness, and the shape of a heart is similar to an upside-down spade, so this could certainly be the case.

For future work we plan to train the model on different distribution of sets (e.g. cards on different background, themed decks of cards etc.). However, as our initial research has shown, it is not trivial to find a large enough training data set that simulates a real-world situation online. We would therefore most likely need to create other data set from scratch by taking pictures, which is a very time consuming exercise. If we had more computational power, we would also try a larger variety of hyper parameters, e.g. to sample a wider variety of learning rates and try to add regularization, as we did with the classical machine learning models, in order to avoid a potential bias in favor of the Hearts suit.

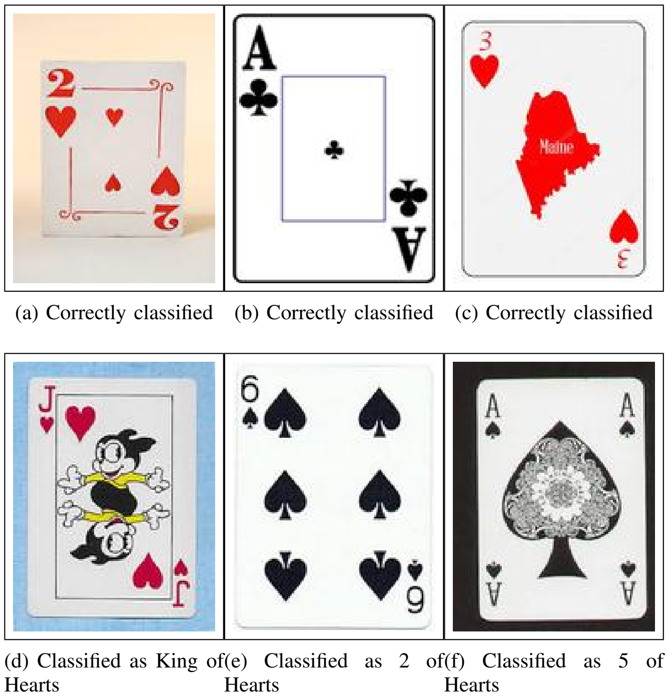

(a) Correctly classified
(b) Correctly classified
(c) Correctly classified
(d) Classified as King of Hearts
(e) Classified as 2 of Hearts
(f) Classified as 5 of Hearts

Fig. 8: Sample of images scraped from internet, both correctly and incorrectly classified

Acknowledgment

This research was conducted as the final project for the course CS229 (*Machine Learning*), at Stanford University. The authors would like to thank Dan Boneh and Andrew Ng, and the rest of the teaching staff. Special thanks are given to the authors project mentor, Sanyam Mehra, for his advice throughout the project.

Contributions

Matias Castillo: worked on the data collection and augmentation, the development of classical machine learning models, and the report.

Benjamin Goeing: worked on the data collection and augmentation, data formatting, the development of the convolutional neural network, and the report.

Jesper Westell: worked on the development of classical machine learning models, in feature extraction and vector manipulation, in testing the convolutional neural network, and the report.

References

[1] Yu, Q., Mao, D., Wang, J. *Deep Learning Based Food Recognition*. 2016. Stanford CS229 Project Report.

[2] DeepLearning.ai / CS230 course materials

[3] https://github.com/grishasergei/conviz

[4] Convolutional Neural Networks (CNNs / ConvNets), http://cs231n.github.io/convolutional-networks/

[5] Zeiler, M. D., Fergus, R. (2014, September). Visualizing and understanding convolutional networks. In European Conference on Computer Vision (pp. 818-833). Springer International Publishing.

[6] Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: NIPS (2012)

[7] TensorFlow. https://www.tensorflow.org/

[8] Scikit-learn. http://scikit-learn.org/stable/

[9] NumPy. http://www.numpy.org/