Sorting Playing Cards: The Rescue of the Deep Convolutional Neural Network

Eren Akgunduz

December 11, 2018 (PSY 4930)

A deck of playing cards, though simple in itself, has already proven its applicability in more complex realms. For instance, consider a standard 52 card playing deck with 13 cards each of 4 different suits. More unique arrangements can theoretically exist from the arrangement of each of these 52 cards than the amount of atoms that exist on Earth – an astronomically large number **(1)**. This completely stochastic distribution of cards lies at the essence of the feasibility for all playing card games. However, one contrasting aspect of playing cards that has not been evaluated to the same extent concerns their organization; more specifically, finding the most efficient method for sorting them by their suit. Some existing approaches to this problem have involved manually programming algorithms in languages such as C++ and Java that would allow a computer to sort a simulated deck of cards – represented only by value **(2, 3)**.

To me, this appeared to be a step in the right direction, since naturally a computer would be faster and intrinsically better suited for a task as mundane as sorting a deck of cards by suit than a human could ever be. That being said, I saw an opportunity to tackle this problem by taking its practicality to the next level. Instead, I would train a deep convolutional neural network using TensorFlow in Python that could receive actual images of playing cards with an array of unique designs and legitimately learn to distinguish between the features of the 4 suits, and what separates a card of the heart suit from a spade, club, or diamond, etc. The ultimate intention is that it

would eventually be able to categorize hundreds, even thousands of images extremely quickly and accurately – thus representing a very effective and innovative solution for this task in incorporating machine learning.

After manually selecting and mass downloading at least 20 images of all 52 playing cards from Google using the Fatkun Batch Download Image extension for Chrome, I eventually collected 1,760 images in total, sorted into 4 subdirectories based on the suit to which the card pictures inside belonged. I coded the CNN in a Colaboratory notebook (.ipynb) through Google Drive, with a GPU enhanced runtime. After importing all the necessary libraries (tflearn, numpy, glob, os, sys, google.colab, scipy.misc) and mounting my data folder into my notebook, I applied an image reading function (courtesy of Michael Teti) which resized my images and appended a categorical label based on the folder they were in (hearts, clubs, spades, or diamonds). Immediately following that, I installed Tensorboard dependencies to visualize the training and began to set up my network architecture. Initially, I was fairly unsure as to what my hyperparameters would be specifically, but from looking at the AlexNet architecture I knew that ReLU activations and at least one fully connected layer would best suit my purpose – in my case, as the final layer with a 'softmax' activation **(4)**. I started Tensorboard and began training my data with three distinct scenarios – two with a validation set ratio of 10% and one with 20%, and one of the 10% validation ratio runs with and one without a snapshot size of 100. Poor performance on my initial architecture led me to remove an excess max pooling and convolution layer, thus leading my final architecture to include: an input layer followed by a convolution layer with 64 7x7 filters with a stride of 2, then a max pooling layer, followed by another

convolution (128 3x3) and max pooling layer, and finishing with a 256 3x3 convolution layer, a global average pool, and a fully connected output layer. Since my data contained 4 classes (suits), I utilized categorical crossentropy in my optimization.

From my Tensorboard graphs, it was clear to see in my first run that the network's validation loss began and remained high in the 16.5-16.6 range, with both validation and aggregate accuracy quickly plummeting to 0% [refer to **Figure 1**]. After revising my network architecture as described above and testing with the same model.fit parameters, however, my validation loss immediately dropped to about/ slightly below 15.7, and with a spiky but steady increase in validation accuracy. After 12 epochs or so, it plateaued at 100%, in stark contrast to my previous training **[Figure 1]**. Upon restarting the kernel/runtime and repeating this iteration identically for the full 50 epochs, the training commenced in a very similar fashion, ultimately finishing with my lowest validation loss yet of 15.68532 and a stable validation accuracy of 1.0 (100%, see **Figure 2**). Tweaking the snapshot size and reducing the validation dataset size only heavily worsened these results, with my final of 3 tests hiking to a validation loss of around 20.3 and plummeting once again to 0% validation accuracy, as shown by cardsCNN3 in **Figure 1**. The following day, I repeated my most successful iteration for the final time and saw a descent to my lowest validation loss yet at 13.5, but with a validation accuracy that consistently fluctuated between 23.86% and 100%, with a training accuracy that peaked at 99.99% and fluctuated in a gradual descent for the remainder of the session, ending at 69.52% **[Figure 3]**.

To create a visualization that could represent this stagnation in order for me to better understand it, I incorporated a confusion matrix. Apart from two regions, the plot

was dark, which I attributed to the quality of the pictures in the validation dataset for that run. Ultimately, though, I am satisfied with my results and elated that the hyperparameters I selected, which seemed generic to me, yielded such a high accuracy at certain points, though I wasn't surprised. I had expected that the convolution and pooling layers would eventually pick up on the defining aspects of the cards and after several observable fluctuations in error be able to categorize all of them, though perhaps in the future architectures such as ResNet and its residual block features could also yield similar results with a lower loss and more consistent validation and training accuracy **(5)**. A future application and extension of the principles I have discussed here that seemed very interesting as it came to mind could involve using GANs to generate new card designs by incorporating a network similar to mine as an evaluator **(6)**. The capacity for deep learning to serve an expansive array of utilities that begins with successfully recognizing categories grows more apparent with the continuing success of experiments similar to the one outlined here.

Figures

Figure 1: All runs

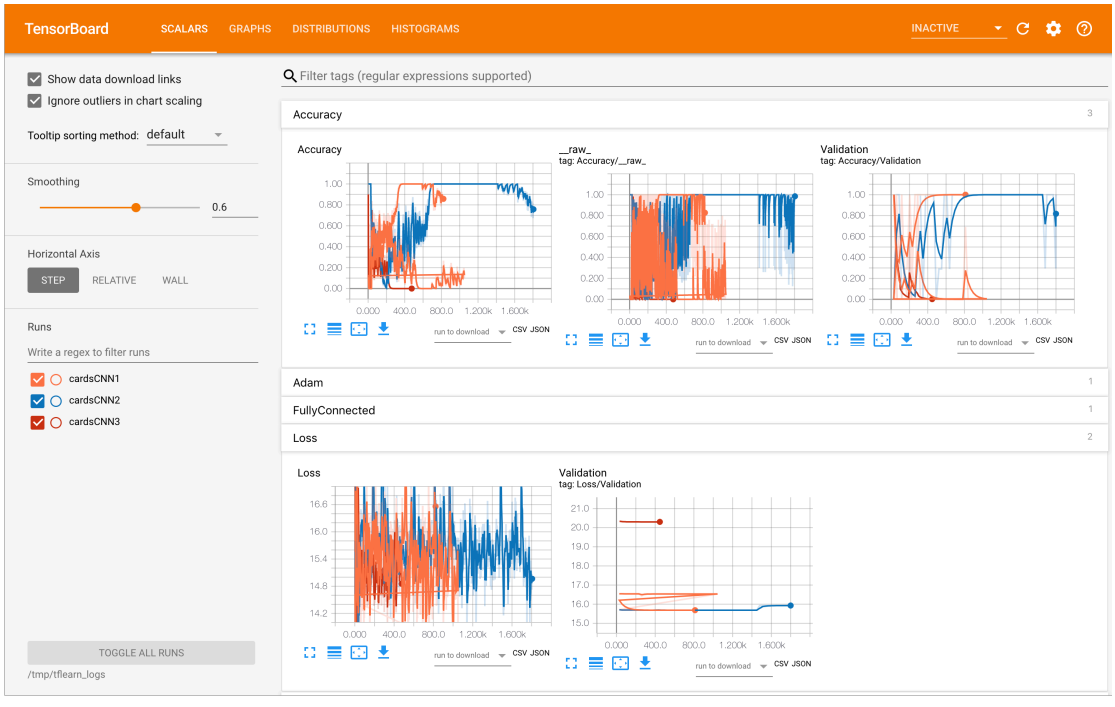


Figure 2: Most successful run

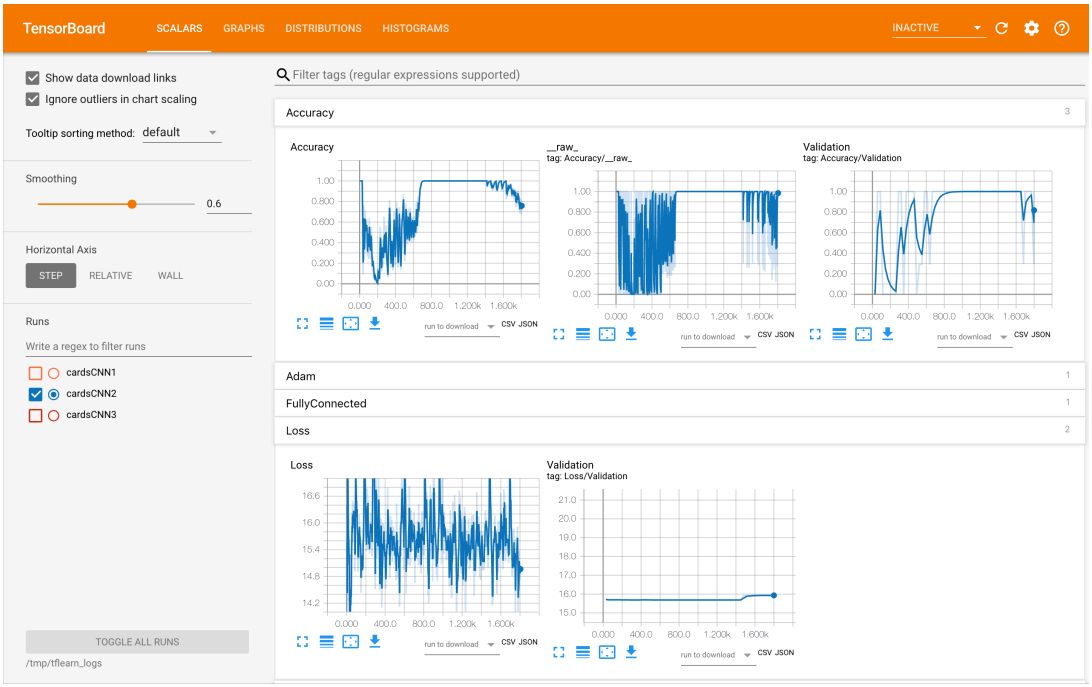
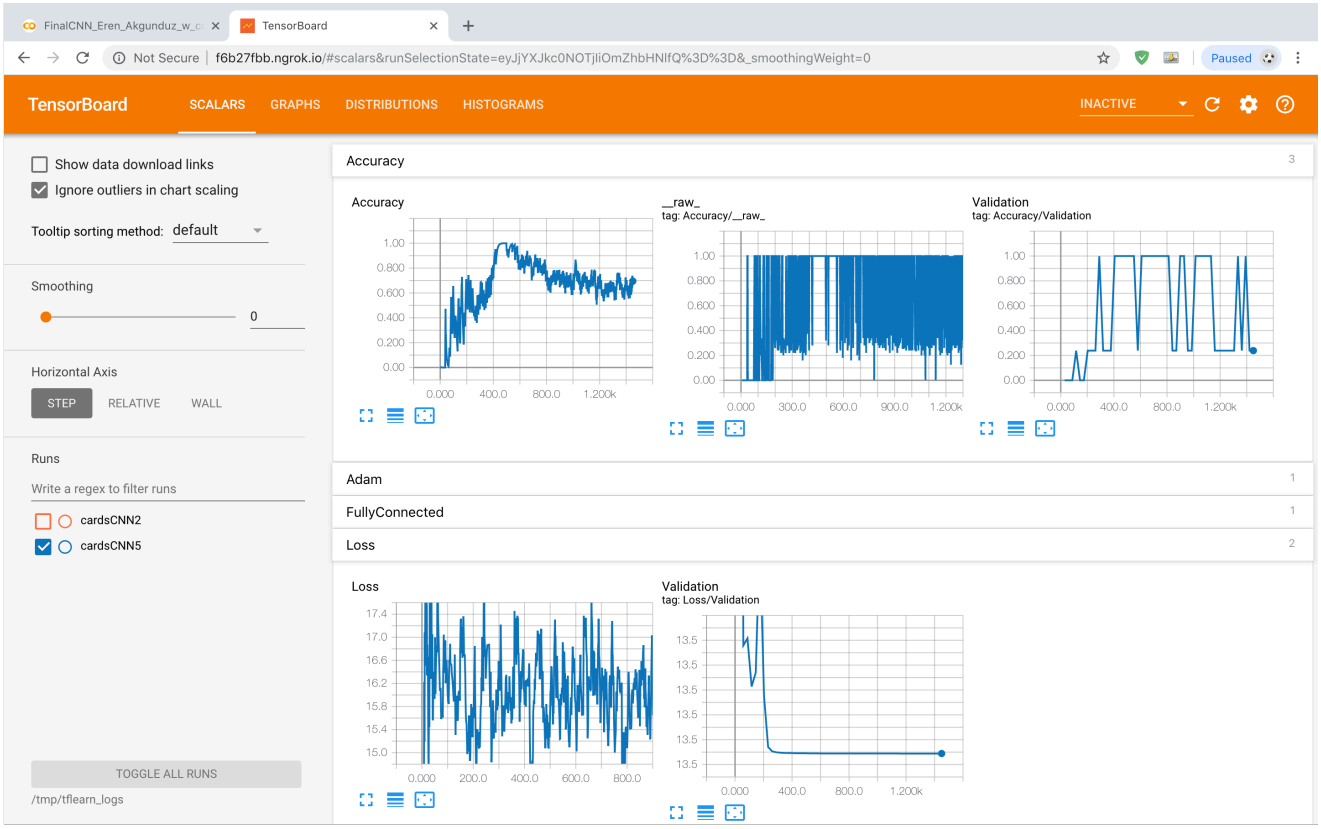


Figure 3: Final run



References

1. https://www.mcgill.ca/oss/article/did-you-know-infographics/there-are-more-ways-arrange-deck-cards-there-are-atoms-earth
2. http://www.cplusplus.com/forum/beginner/219420/
3. https://stackoverflow.com/questions/4650042/sorting-a-deck-of-cards-by-suit-and-then-rank
4. https://github.com/mpcrlab/DeepLearningFall2018/blob/master/Papers/AlexNet.pdf
5. https://github.com/mpcrlab/DeepLearningFall2018/blob/master/Papers/ResNet.pdf
6. https://www.toptal.com/machine-learning/generative-adversarial-networks