Tutorial Report

Anthony Vasquez

1 Methods

What I accomplished for this tutorial was making an image classification system using a Convolutional Neural Network (CNN). I followed along with a YouTube tutorial, which I coded line by line and ran the current code block each time.

For the prerequisites, tensorflow and keras were installed, as these libraries are satisfactory for pulling an existing dataset and building the neural networks (NN). Numpy was also installed, since images would needed to be converted to a numpy array for the neural network to read the image pixel values. The dataset chosen was images with different images showing the hand poses for rock, paper, or scissors. This dataset is apart of tensorflow's dataset catalog, which makes collecting these kinds of images on my own unnecessary for training and testing.

The next segment of work involved looking at what data I am working with, and how to prepare the images as input for the neural networks. Foremost, I used a built in tensorflow function that provides the metadata of the rock, paper, scissors (RPS) dataset. The relevant information I needed was the amount of images for training and testing, as well as the image and label features. Moreover, I used another function to show what these images actually looked like, which each image has a pose and a label of what the pose is. Afterwards, I split the training and test images into their own variable placeholders to make selecting either category simpler for selection. The image and label features for the training and test dataset were converted into a numpy array, as well as changing the data type of the pixel values into a float. The converted float values were normalized into a value between 0 and 1 by dividing all pixels by 255, as it is standard for a pixel value to hold an RGB value between 0 and 255.

Before building a CNN, a basic neural network was built first using the keras library. The NN was declared as a **Sequential model**, which different layers were coded in. The first layer was the **Flatten layer**, which converted the numpy array of the image input into a single column. For this network, it was necessary to change the dimensions of the array for the preceding layers to work with the numpy array. The next two layers were **Dense layers**, which each layer was given a size and an activation function to process the input. The final layer was a dense layer, but this serves as the output layer, which had a size of 3 (3 possible poses of RPS) and a different activation function for determining the most likely pose. A loss optimizer was included, which tweaks the learning and accuracy of the NN. I ran a training session of the built NN, which took in the training dataset and given 5 **epochs** (execution intervals), as well as a **batch size** of 32 (i.e train for every 32 images in the dataset). After viewing the training results, I ran an evaluation test of the NN on the test dataset.

The next NN to create was a Convolutional Neural Network, which specific layers were created just like with the NN. The CNN was declared as a Sequential model as well. The first two layers were **convolutional layers**, which each layer performs its own convolutions by sweeping a 3x3 "grid" on a pixel group of the image pixels to extract its features (e.g edges). The preceding layer flattens the results of these convolutions, which then go to the output layer. The same loss optimizer was used, as in the NN. I ran the same training dataset on the CNN to compare the accuracy results with the first NN. However, to run the training for the CNN I switched the environment of Google Colab to run on a GPU to speed up runtime execution.

To improve the accuracy of using a CNN, additional layers were added with the ones before for a new CNN. An **average pooling layer** was added as the first layer, which a 6x6 "grid" is swept across the image but the average of the pixels is computed to be the new representation of the image at each sweep. This reduces the image resolution enough to still maintain the important features for the other layers to perform convolutions on. A **Max Pooling layer** was added with a similar purpose as the average pool layer. Essentially, a smaller grid sweeps across the image, after its pixel size is downsized from the average pool layer, and it finds the highest pixel value within each grid sweep. This layer refines the convolution further in extracting out image features, as things such as edges will have distinct pixel values that the Max Pooling layer may pick up on. A **Dropout layer** was added after, which drops a percentage of the dataset images in the model's runtime. This layer means to randomize what images the CNN sees as to further generalize its performance output, which helps against the overfitting problem.

In configuring the layers, it helps with finding the optimal parameter values, which is was the goal of using Kerastuner for the CNN model. With Keras Tuner, there are different function calls that can be used within each layer to try combinations of different parameter values. From reading the

documentation, I used the *hp.Choice* function call on a normal convolution layer, which provided different choices of filter sizes for the convolution grid to perform. Running the Keras Tuner with just this one chosen layer modified, a trial session ran, which different filter values were tested individually against the test dataset. The output is the parameter value for the filter size that was deemed the most optimal. I performed this same combination trial of parameters on a couple of other layers, as to observe if the model did perform better if more layers were added in choosing optimal parameter values concurrently.

2 Evaluation Metrics

The only metric used for image classification was accuracy, which determined if the neural network identified the right pose from the RPS game. When the model was trained, the accuracy metric ranged between 0 and 1, which was used for both the training and testing runtimes. The accuracy measurements were used as an indicator for improving upon a NN with a CNN, and then modifying the CNN with additional layers. This was the only metric used, as it was a simple evaluation to to test the models against. There were no other metrics defined to prompt further tweaking of building the CNN.

3 Results and Discussions

This project was insightful in creating a basic CNN for image classification, but the results are not as in depth. Rather, the focus of the tutorial was on the learnings of implementing a CNN for image classification. The beginning part of the tutorial that focused on the data preparation of the images was very interesting because I did not expect the technical structure and conversion of our images to input for our neural networks would take this much effort. I figured that the image could be inputted as the actual image file itself and then proceed with making the NN. However, hearing the instructor explain the necessity to convert the images to a numpy array, as to represent the pixel values, and also normalizing the pixel values for better performance was helpful in understanding the nuances of inputs for a NN.

Another remark of the tutorial was the progression of using a CNN and modifying it. At first, a simple NN was built for image classification, and the training results seemed promising with a near 90% accuracy. However, once the test dataset was evaluated, it was apparent that this simple NN was not as proficient when it came to unknown images it hasn't seen. Essentially, the NN was accurate for the training dataset because it overfitted for the images it already trained on. Turning to a CNN to dampen the overfitting problem was interesting because I understood that the goal of achieve accurate classifications can really be measured when the model we use can generalize in its learning of images. As such, the explanation of how a CNN works with "sweeping" the image for features using a smaller grid (i.e the filter) helped a lot in understanding the different layers that make up a CNN.