CAP4453 – Spring 2019 Justin Powell
PA3 Report

Goal:
For PA3, we were tasked with creating an image classifier that recognizes actions from the UCF sports action dataset.

Dataset and Data Loader:
The dataset used in PA3 consisted of 10 actions with accompanying videos and video frames. The actions included in this dataset were *diving, golf swing, kicking, lifting, horse riding, running, skateboarding, swing-bench, swing-side* and *walking.*

Originally the data was organized as *ucf_actions/action/number/image.jpg* where the action iterated through all 10 previously listed actions and the number iterated through the amount of videos underneath that action. I decided to reorganize the data to aid in efficiently loading it so that I could test and train the data. I split the dataset into a training dataset and a testing dataset, where 70% of the original data went into a folder named *train* and 30% of the original data went into a folder named *test*. From there, I created folders named after each action and placed the data under the respective action; thus, reorganizing the data into a *train/action/image.jpg* and *test/action/image.jpg* structure.

Reorganizing the data in this structure was greatly beneficial when loading the data. Using a *root/class/image* structure allowed me to use the Image Folder class that is offered by pytorch. Image Folder loaded the data for me, used the folder name as the class name and tied each image to its corresponding class. I transformed the data by first resizing it to a 32x32 for consistency, then turning it into a tensor and lastly normalizing the data. From there I was able to use the Dataset class which represents the loaded data as an actual dataset. I loaded both the test and train data to create a testing and training dataset, respectively.

Classifier Design and Feature Extraction:
For our task to create an image classifier, I decided to create a Convolutional Neural Network, or CNN for short. This CNN consisted of two convolutional layers, two pooling layers, an ReLU activation function, two fully connected layers, and 1 hidden layer. After testing and restructuring the CNN, I decided to choose two convolutional layers after introducing more layers resulted in higher computation costs despite giving no significant increase in accuracy. I decided to model the fully connected and hidden layers after our PA2 assignment. Feature extraction was achieved by forwarding our input, or image, through our forward function.

The forward function is the directional sequence of all the layers in the CNN. The first layer is the convolutional layer which accepts a 3-dimensional input, outputs 20 features, using a stride of 1, pads with 1 and convolves with a 5x5 kernel size. Forwarding the input through this layer, we get the first batch of features which is activated with an ReLU function and sent through a pooling layer with a 2x2 kernel size and a stride of 2. The features from this layer are then sent through another convolutional layer to be used as input. This convolutional layer repeats the previous process and extracts 40 features, which are then flattened down and sent to a fully connected layer. The fully connected layer takes a flattened input dimension of 40 * 6 *6 and attempts to extract 100 features. The reasoning behind this large difference in input and output is to, in theory, prune the data and extract the more significant features. This process is repeated again with another fully connected layer, then running the 100 features to a hidden layer which finally outputs 10 features.

Evaluation:
During the process of training the model several times, I did various things to optimize the model and increase accuracy. More specifically, when it came to the structure of the CNN, I added and/or reduced the number of layers in the model. After finding that 2 convolutional layers and 3 fully connected layers worked best for me, I then started to make changes to the loss function and optimizer used during training. My first version of the CNN was using an SGD Momentum optimizer and an NLL loss function that were both implemented in PA2. This optimizer had a learning rate of .1, a momentum of .9 and a weight decay of .00001. Using this optimizer with the mentioned parameters resulted in a loss of accuracy over the 20 epochs as well as a divergence of the loss function and an overflow in the lost value. After much research, I concluded that this loss divergence stemmed from the learning rate. After decreasing the learning rate to .003 to accommodate to our larger dataset, the resulted in the loss function converging and decreasing at a slower rate and showing an accuracy of 19.25% over 20 epochs.

To optimize this further, I looked into completely changing the optimizer I was using from an SGD to an Adam optimizer. I once again made a slight decrease to the learning rate, passing .001. Since Adam does not require momentum, the only other parameter needed was a weight decay, which did not change from above. Additionally, I changed the loss function from an NLL to a Cross Entropy, which is actually the combination of both the Log SoftMax and NLL functions. I also reduced the batch size of the test set from 100 to 50. With the introduction of both a new optimizer, a new criterion and a reduced batch size, my CNN model ended in an accuracy of 46.58% over 20 epochs.

Despite an accuracy of 19% in the first model and increase to 46% in the second model, both models did not seem to be stable. Over the 20 epochs, there were fluctuations in accuracies; the first fluctuating between ~19% and ~27% and the second fluctuating between ~40% and ~48% (the accuracies are showcased in the *CNN Outputs* folder). To further test stability, I increased the epoch from 20 to 30 which resulted in a final average accuracy of 38%. Running the test set through a saved copy of my model also resulted in an accuracy of 38%. To overcome this stability issue, I believe a possible solution could be to increase the epoch to around 100 to accommodate for the batch size of 50 I used. Additionally, this solution may also increase the model's accuracy.

Conclusion:
To conclude, action recognition was achieved by training a convolutional neural network on 70% of the original data and then testing that model on the remaining 30%. Using an Adam optimizer and a Cross Entropy criterion I was able to achieve an accuracy of 38%, albeit inconsistent. Despite said inconsistency, I gained a lot of experience through this challenging assignment and am proud that I was able to construct my very own image recognition model.