ECSE 4850 Final Project

Training a deep neural network to recognize human actions in video data

---

Note: For information regarding loading and evaluating the model, see the end of this report.

Problem Statement

For this project, our task was to build and train a deep neural network to recognize human actions in video data. Given some input sequence of frames, the model should be able to classify the video into one of eleven classes based on the action that the human is performing in the video. We are given preprocessed data from the UCF11 YouTube Action dataset for training and validation of the model, but the actual separation ratio is not fixed. The structure of the model is not fixed either, and it is up to us to determine an effective model architecture and implement it. Additionally, model hyper parameters are not fixed and we are free to use any techniques to combat overfitting as needed. Full usage of the Tensorflow library is allowed and therefore no gradient computations need to be done by hand.

Environment Settings

For development of this project, an Anaconda virtual environment was created with Python 3.8 and Tensorflow 2.3.0, and programming and debugging was completed using the Spyder IDE. To improve model training speed, Tensorflow was connected with my system's GPU via CUDA version 10.1 and CuDNN version 7.6. For reference, my system has an Intel Core i7-7700HQ 4-core CPU, an NVIDIA GeForce GTX 1050 GPU with 4GB of dedicated memory, and 16GB of physical RAM.

Model Architecture

Given the complexity of the task, I started by creating a model architecture with a convolutional neural network to capture image features followed by an LSTM layer to capture temporal dependencies. My original model followed the VGG-16 architecture with an LSTM layer at the very end. However, when training the model, I noticed that the model would exhibit extreme overfitting after just two epochs, and so I significantly decreased the complexity of the model. My final architecture is described in more detail below. (note that "Same" padding indicates that padding was applied to the input such that the dimensions of the output match the dimensions of the input)

	Name	Layer	Window	Filters/Units	Regularization	Padding	Activation
1	`conv1`	Conv2D	3×3	32 filters	L2	Same	ReLU
2	`max1`	MaxPool2D	2×2, stride 2	-	-	-	-
3	`conv2`	Conv2D	3×3	32 filters	L2	Same	ReLU
4	`max2`	MaxPool2D	2×2, stride 2	-	-	-	-
5	`conv3`	Conv2D	3×3	64 filters	L2	Same	ReLU
6	`max3`	MaxPool2D	2×2, stride 2	-	-	-	-
7	`conv4`	Conv2D	3×3	128 filters	L2	Same	ReLU
8	`conv5`	Conv2D	3×3	128 filters	L2	Same	ReLU
9	`max4`	MaxPool2D	2×2, stride 2	-	-	-	-
10	`lin`	Dense	-	256 units	L2	-	ReLU
11	`lstm`	LSTM	-	128 units	-	-	-
12	`out`	Dense	-	11 units	L2	-	Linear

Table 1: Detailed Model Architecture

The final layer of the model has no activation because the model loss was computed using `tf.nn.softmax_cross_entropy_with_logits()`, which will compute the softmax automatically before comput-

ing the loss (for increased numerical stability). Additionally, to combat overfitting, the following improvements were made:

- *Weight Regularization*: All layers of the model that had weights also had L2 regularization applied to their weights. This was used to penalize large weight values and therefore prevent overfitting.
- *Learning Rate Decay*: For training, the Adam optimizer was used, and therefore the learning rate for each layer's weights were reduced and changed over time via the Adam algorithm.
- *Data Augmentation*: To reduce the possibility of overfitting even further, with probability of 50%, all frames of a certain data sample would be horizontally flipped. This encourages the model to learn important features and to not simply memorize the training data set.

Hyper Parameters

- *Data Split Ratio*: For this assignment, the data split ratio was set to 0.75. To clarify, that means that the training data set contained 75% of the total number of data samples available, and the validation set contained 25%.
- *Optimizer*: For this assignment, the Adam optimizer was used.
- *Learning Rate*: The initial learning rate was set to 0.0001. I initially set it to 0.001, but that ended up being too large for the Adam optimizer to be effective.
- *Weight and Bias Initialization*: All weights were initialized to a small random value from a normal distribution, i.e. $W \sim N(0, 0.05)$. All biases were initialized to 0.
- *Batch Size*: The batch size was set to 8. I found that smaller batch sizes resulted in less effective learning, and my GPU's memory could not handle a larger batch size.
- *Number of Epochs*: The model was set to train for 25 epochs.

Pseudocode

```
load training and validation data sets
normalize video frames and create one-hot label vectors
initialize model
create Datasets from data generator functions

for epoch = 1 to 25
    for each batch in training data set
        compute predictions
        compute loss and accuracy
        compute gradient and update weights
    end loop

    for each batch in validation data set
        compute predictions
        compute loss and accuracy
    end loop

    aggregate loss and accuracy values and compute metrics for this epoch
end loop

plot average loss, overall accuracy, and class-wise accuracy
create confusion matrix heatmap
save model
```

Model Performance

Training & Validation Loss

The model was trained on 5,448 videos (75% of the total 7,272 samples) and validated on 1,824 videos (25% of the total 7,272 samples). As the model was training, its training and validation loss values were recorded and plotted. The average loss of the model for each of the 25 epochs (the average of the losses computed for each batch of image sequences) is shown in the plot below.

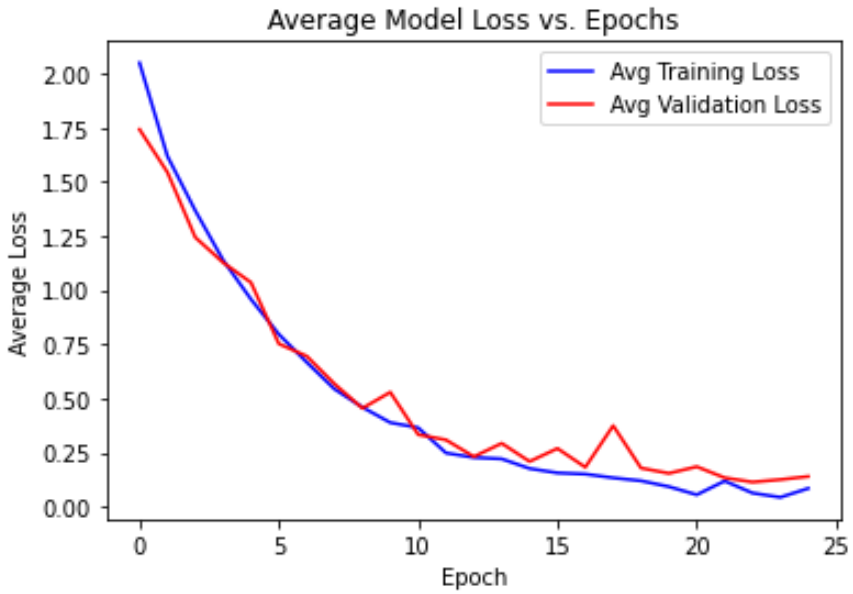

Figure 1: Training and Validation Loss as a Function of Epochs

The above plot conveys many important properties of the model as it was training:

1. Both the training and validation loss decreased very quickly at first and then continued decreasing but at a much slower rate. This is the expected behavior.

2. The training and validation loss curves follow each other very closely, which is a good indication that the model is not overfitting. There are a few points where the validation loss spiked, but it stabilized and returned to the training loss curve by the next epoch. Towards the end of the training process, it looks like the training loss curve deviates from the validation loss curve very slightly, which may indicate that the model will overfit if training continued.

3. It seems that both loss curves are just starting to stagnate which may be an indication that training the model for a few more epochs could be beneficial. If training continued, I would have to watch very closely for overfitting.

At the end of training, the final training loss value was 0.09 and the final validation loss value was 0.14.

Training & Validation Accuracy

The accuracy of the model was evaluated during training as well. The training and validation accuracy of the model is plotted in the figure below. Note here that "accuracy" is defined as the ratio of correct classifications to the total number of images.

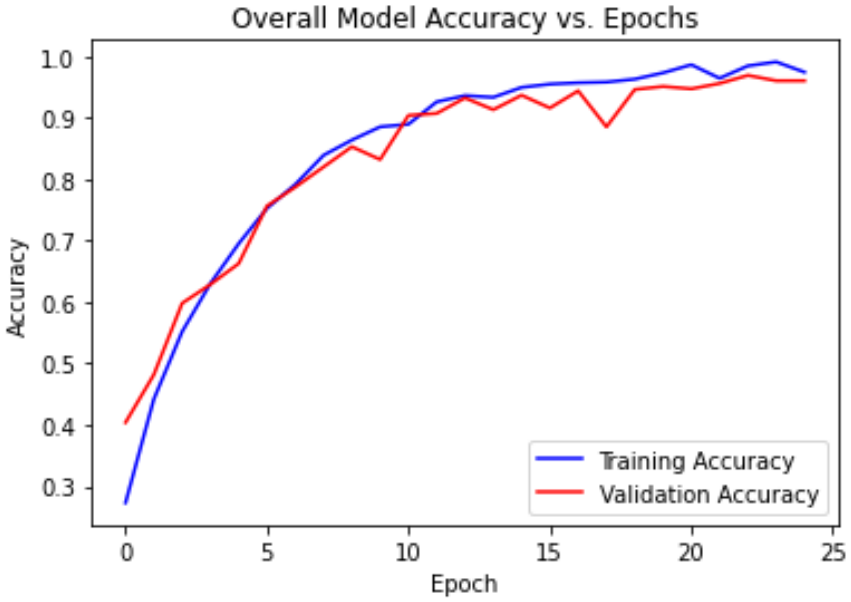

Figure 2: Training and Validation Accuracy as a Function of Epochs

The above plot shows that the accuracy of the model increased steeply at first, and then started to increase more slowly as more epochs occurred. This is expected and indicates that my implementation is likely correct. Additionally, similar to the loss curves, the training and validation accuracy curves follow each other very closely, which is a good indication that the model is not overfitting. It is interesting to note that these curves closely resemble the loss curves if they were inverted, and therefore the observations made for the loss curves also apply here. At the end of training, the final training accuracy was 97% and the final validation accuracy was 96%. The validation accuracy for each class is summarized in the table below.

	Class	**Abbreviation**	**Accuracy**
0	Basketball Shooting	`b_shooting`	97.37%
1	Cycling	`cycling`	84.02%
2	Diving	`diving`	98.34%
3	Golf Swinging	`g_swinging`	100%
4	Horseback Riding	`h_riding`	92.70%
5	Soccer Juggling	`s_juggling`	98.91%
6	Swinging (on a swingset)	`swinging`	96.86%
7	Tennis Swinging	`t_swinging`	99.47%
8	Trampoline Jumping	`t_jumping`	98.54%
9	Volleyball Spiking	`v_spiking`	98.29%
10	Walking with a Dog	`d_walking`	93.66%
	Overall		**96.05%**

Table 2: Class-wise Validation Accuracy Values

It is difficult to know exactly what the model is thinking when classifying, but the above table can give us some insight. It appears that the model has the most difficulty with identifying cycling, horseback riding, and walking

with a dog. It also appears that it has the least difficulty with identifying golf swinging and tennis swinging, likely due to the distinct surroundings that both sports involve. Overall, the model has relatively the same accuracy for each class (except for cycling) which is a good indication that the model is learning important features and the proper temporal dependencies to differentiate between classes. For reference, the accuracy values for each class have been plotted over time in the figure below. Each of the plots show that the training and validation accuracy values decreased over time. (Note: the plots in the figure below may have different y-axis scales.)

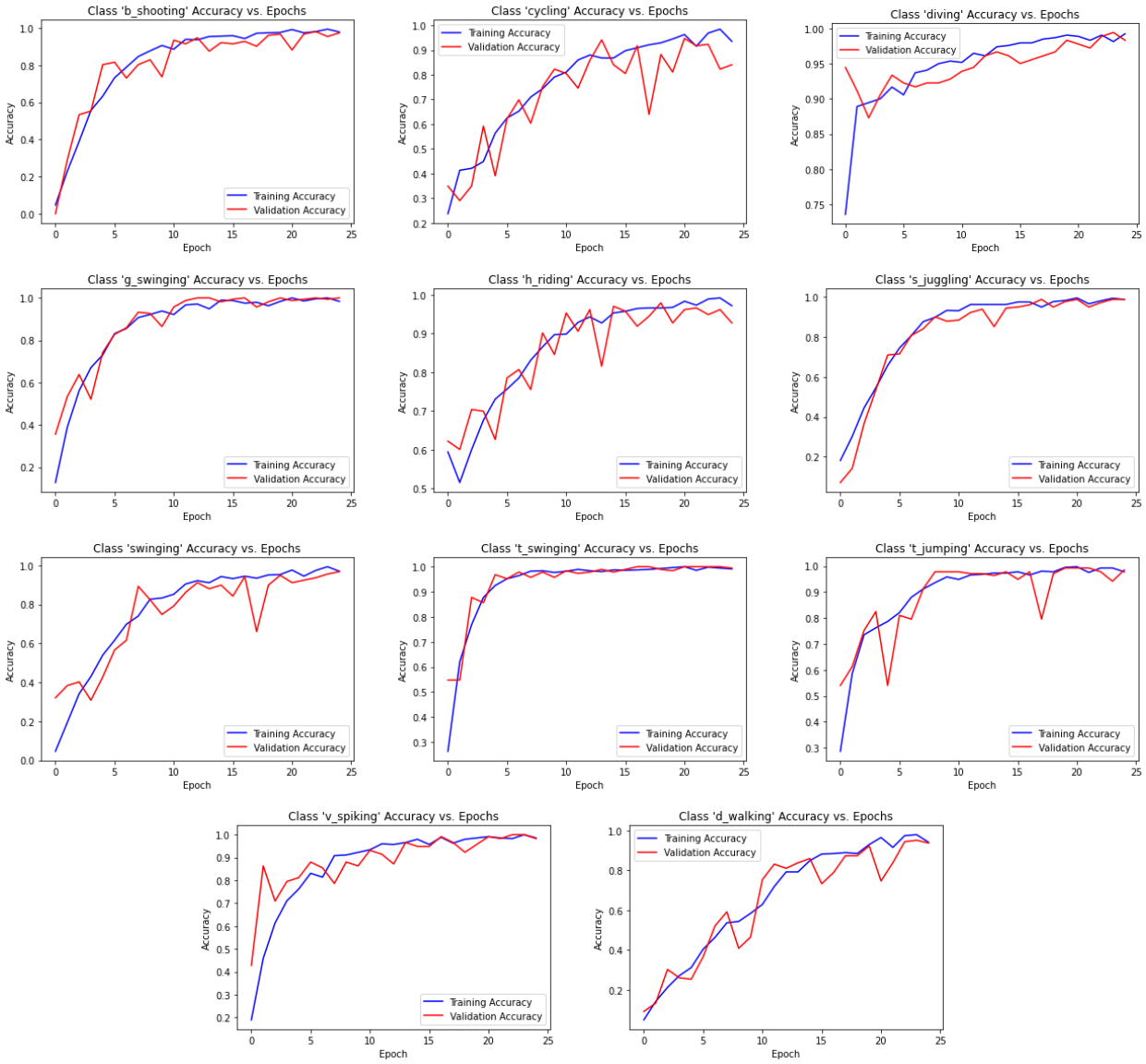

Figure 3: Class-wise Training and Validation Accuracy as a Function of Epochs

Confusion Matrix

To further illustrate the model's performance when classifying the validation data set, we can build a confusion matrix. This figure has the true labels on the y-axis and the predicted labels on the x-axis. Each entry (x, y) in the matrix shows the number of samples with true label y that were predicted to have a label of x. If the model were 100% accurate, the matrix would have high numbers on the diagonal and 0 elsewhere, indicating that the model had no incorrect predictions. The figure below shows a heatmap of the confusion matrix for this model when evaluated on the validation data.

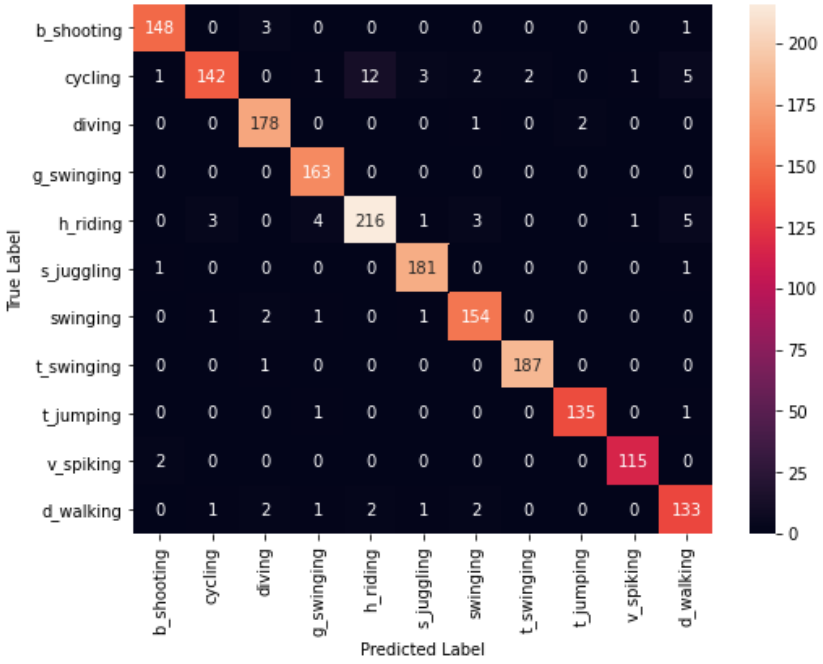

Figure 4: Confusion Matrix with Validation Data

The confusion matrix shows that, for the most part, the model is highly accurate. We have very large numbers on the diagonal and a few small non-zero entries elsewhere.

Conclusion

In conclusion, I was able to successfully build and train a model to recognize human actions in video data. The model's final accuracy on the validation data was 96%, which is exceptionally good given the complexity of the task. Similar to the previous assignment, I was limited by my low amount of GPU memory, which meant that my model took a longer amount of time to train. Despite this, I was able to find a model architecture that was complex enough to learn the important features and temporal dependencies in the videos but simple enough to run smoothly on my machine. I also learned a lot about how simplifying the model architecture can be a very effective way of combating extreme overfitting.

Future Considerations

Unlike the other assignments, I don;t have many improvements that I would make to the model. The final performance of this model was very good (in my opinion) and therefore there are not many improvements that can be made. That said, the last few percentage points of accuracy might be attainable by slightly increasing the model complexity. This increase in complexity may also increase the likelihood of overfitting though, and therefore more regularization techniques will need to be applied, for example, adding dropout layers or adding more data augmentation.

Final Note

I want to give a huge thanks to Professor Ji and TAs Xiao and Naiyu for all of their hard work this semester. I will admit the class has been quite difficult, but I have really enjoyed taking it and I learned a ton. I'm sure I will remember the content of this class for years to come, and that is in part due to the contributions from the professor and the TAs. Be well, and I hope you all enjoy your summer.

Model Loading & Evaluation

The model I created was built by subclassing the `tf.keras.Model` class. As such, it was saved using the standard Keras model saving process (`model.save("DeepClassifierModel")`) which saves the model in the `SavedModel` format. However, the model was not trained using the standard Keras `model.fit` method, and therefore it cannot be evaluated using the normal `model.evaluate` method. Included in the submitted .zip archive is a Python file named `run_file.py`. This file can be used to evaluate my model's performance on a given set of testing data as follows (this information is also available in the `README.txt` file):

1. At the top of the file, change the values for the parameters to match the locations of the files needed:
    - `TEST_DATA_PATH` should have the path to the testing videos .npy file. This data should be non-normalized, i.e. each pixel is an integer in the range [0, 255].
    - `TEST_LABEL_PATH` should have the path to the testing labels .npy file. The labels should be integers in the range [0, 10] (the one-hot vectors will be built when the program is run).
    - `MODEL_PATH` should have the path to the folder containing the saved model to be evaluated. This will likely not need to be changed.
    - `BATCH_SIZE` should be an integer value specifying the batch size for evaluation. My machine's memory is not very large, so this was set to 8 for my testing. However, your machine may be more powerful, so you can increase this value to decrease the time it takes to evaluate the model.
2. Run the python file.
3. The file will print the model's average loss, overall accuracy, class-wise accuracy, and will create a confusion matrix heatmap figure.

Please let me know if you have any issues with evaluating this model!