ECSE 4850 Programming Assignment 4

Traning a CNN-LSTM-MLP model to identify joint positions in human poses

---

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

Problem Statement

The objective of this programming assignment is to train a model to determine 3D human joint positions in a given video (sequence of images/frames). The model must contain a Convolutional Neural Network (CNN) module, an LSTM module, and a Multi-Layer Perceptron (MLP) module, in that order. The specific architecture of the model is not specified, but there must be a CNN, an LSTM, and a MLP. Additionally, hyper parameters are not fixed; the learning rate, batch size, and number of epochs can all be changed as needed. For this assignment, full usage of the Tensorflow library is allowed, and therefore no gradient computations or weight update 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 Description

CNN Architecture

The CNN module of this model was inspired by the structure of the standard VGG-16 network, which is known to have very good performance when classifying images. The CNN implemented in this assignment, however, is scaled down so that my system can handle the training and evaluation steps without running out of memory. The specific details are as follows (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	Padding	Activation
1	`conv1`	Conv2D	3×3	64 filters	Same	ReLU
2	`conv2`	Conv2D	3×3	64 filters	Same	ReLU
3	`max1`	MaxPool2D	2×2, stride 2	-	-	-
4	`conv3`	Conv2D	3×3	128 filters	Same	ReLU
5	`conv4`	Conv2D	3×3	128 filters	Same	ReLU
6	`max2`	MaxPool2D	2×2, stride 2	-	-	-
7	`conv5`	Conv2D	3×3	128 filters	Same	ReLU
8	`conv6`	Conv2D	3×3	128 filters	Same	ReLU
9	`max3`	MaxPool2D	2×2, stride 2	-	-	-
10	`conv7`	Conv2D	3×3	256 filters	Same	ReLU
11	`conv8`	Conv2D	3×3	256 filters	Same	ReLU
12	`max3`	MaxPool2D	2×2, stride 2	-	-	-
13	`conv9`	Conv2D	3×3	256 filters	Same	ReLU
14	`conv10`	Conv2D	3×3	256 filters	Same	ReLU
15	`max4`	MaxPool2D	2×2, stride 2	-	-	-
16	`lin1`	Dense	-	1024 units	-	ReLU
17	`lin2`	Dense	-	1024 units	-	ReLU

Table 1: Detailed CNN Architecture

Additionally, batch normalization was applied after each MaxPool2D layer to improve model accuracy.

LSTM Architecture

The LSTM module of this model is very simple and consists of a single Keras LSTM layer with 512 units.

MLP Architecture

The MLP module of this model consists of a few fully-connected neural network layers, the details of which are described below.

	Name	Layer	Units	Activation
1	`mlp1`	Dense	512	ReLU
2	`mlp2`	Dense	256	ReLU
3	`mlp3`	Dense	51	Linear

Table 2: Detailed MLP Architecture

The output of the last Dense layer in this MLP module is reshaped to be 17×3 which represents the 3 coordinates (x, y, z) of each of the 17 joints.

Hyper Parameters

- *Optimizer*: The basic SGD optimizer was used for this assignment. I found that using the Adam optimizer resulted in too great of a decay in the learning rate, and the model would not improve.
- *Learning Rate*: The learning rate was set to 0.01 for this assignment. No learning rate decay was used in order to maximize the model's accuracy gains without having to let the model train for multiple days.
- *Weight and Bias Initialization*: All weights and biases for each layer were initialized using the default Glorot Uniform initializer, where values are samples from a range of $[-limit, limit]$, where

$$limit = \sqrt{\frac{6}{(fan\_in + fan\_out)}}$$

  and fan_in is the number of input units in the weight tensor and fan_out is the number of output units.
- *Batch Size*: The batch size was set to 3, since larger values would cause my system to run out of memory.
- *Number of Epochs*: The model was set to train for 20 epochs.

Loss Function & Accuracy Equations

Loss Function Equations

The loss function used for this assignment was the Mean Squared Error loss function, a common choice for regression problems such as this one. In general, the loss function is as follows, given M data samples, a prediction $\hat{y}[m]$, and a label $y[m]$:

$$l(y[m], \hat{y}[m]) = \frac{1}{M} \sum_{m=1}^{M} (\hat{y}[m] - y[m])^2 \tag{1}$$

Below, we express equation (1) more specifically in terms of this assignment, given M training samples, N frames per sample, J joints to identify, a prediction $\hat{y}^{M \times N \times J \times 3}$, and a label $y^{M \times N \times J \times 3}$:

$$l(y[m], \hat{y}[m]) = \frac{1}{M \cdot N \cdot J \cdot 3} \sum_{m=1}^{M} \sum_{n=1}^{N} \sum_{j=1}^{J} \sum_{d=1}^{3} (\hat{y}[m, n, j, d] - y[m, n, j, d])^2 \tag{2}$$

In practice, equation (2) is computed per batch and then averaged over all batches in the epoch, which yields the same result.

Mean Per Joint Position Error Equation

The accuracy of the model was evaluated during training as well by measuring the Mean Per Joint Position Error (MPJPE). MPJPE is calculated as the average euclidean distance in millimeters between the predicted position and the ground truth position over all 17 joints, all 8 frames per video, and all videos in the batch. More specifically, given M training samples, N frames per sample, J joints to identify, a prediction $\hat{y}^{M\times N\times J\times 3}$, and a label $y^{M\times N\times J\times 3}$:

$$MPJPE(y[m],\hat{y}[m]) = 1000\cdot\frac{1}{M\cdot N\cdot J}\sum_{m=1}^{M}\sum_{n=1}^{N}\sum_{j=1}^{J}\sqrt{\sum_{d=1}^{3}(\hat{y}[m,n,j,d]-y[m,n,j,d])^2} \tag{3}$$

We multiply by 1000 to convert the measurement to millimeters (mm). In practice, equation (2) is computed per batch and then averaged over all batches in the epoch, which yields the same result.

Pseudocode

```
load training and validation data sets
normalize video frames
initialize model
create Datasets from data generator functions

for epoch = 1 to 20
    for each batch in training data set
        compute predictions
        compute loss and MPJPE
        compute gradient and update weights
    end loop

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

    compute average loss and MPJPE for this epoch
end loop

plot average loss and MPJPE
save model
```

Model Performance

Training & Validation Loss

The model was trained on all 5,964 videos in the training set and validated on all 1,368 videos in the validation set. 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 20 epochs (the average of the losses computed for each batch of image sequences) is shown in the plot below.

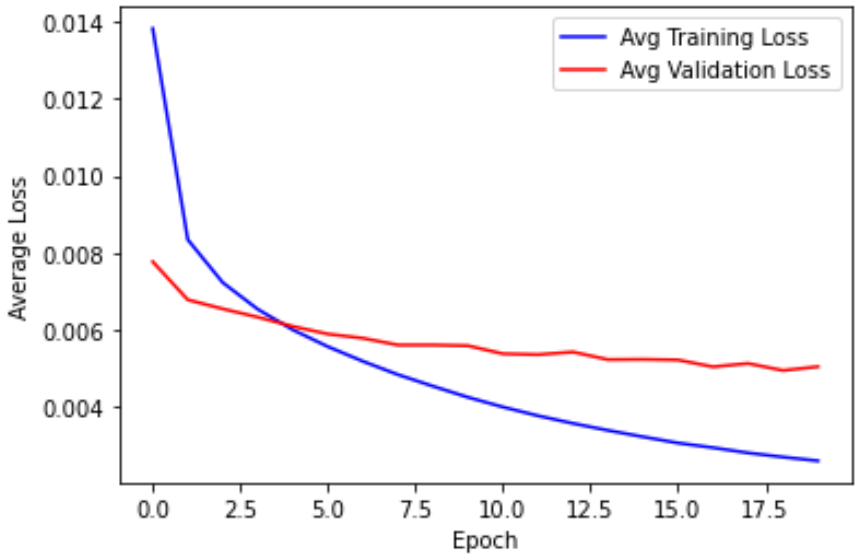

Figure 1: Training and Validation Loss as a Function of Epochs

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

1. The training loss of the model decreased very quickly at first and then continued decreasing but at a much slower rate. The validation loss of the model also decreased very quickly at first and then continued decreasing but at a much slower rate than that the training loss.

2. As more epochs were processed, the training loss and validation loss deviated more. This is a strong indication that the model began to overfit, which is expected given the complexity of the model.

3. It seems that both the training loss and validation loss have not stagnated yet, indicating that training the model for more than 20 epochs would yield higher accuracy. This also may be partly a result of the lack of a learning rate decay.

At the end of training, the final training loss value was 0.00259 and the final validation loss value was 0.00504.

Training & Validation Accuracy

The accuracy of the model was evaluated during training as well by measuring the Mean Per Joint Position Error (MPJPE). MPJPE is calculated as the average euclidean distance in millimeters between the predicted position and the ground truth position over all 17 joints, all 8 frames per video, and all videos in the batch. The average MPJPE for each of the 20 epochs (the average of the MPJPE computed for each batch of image sequences) is shown in the plot below.

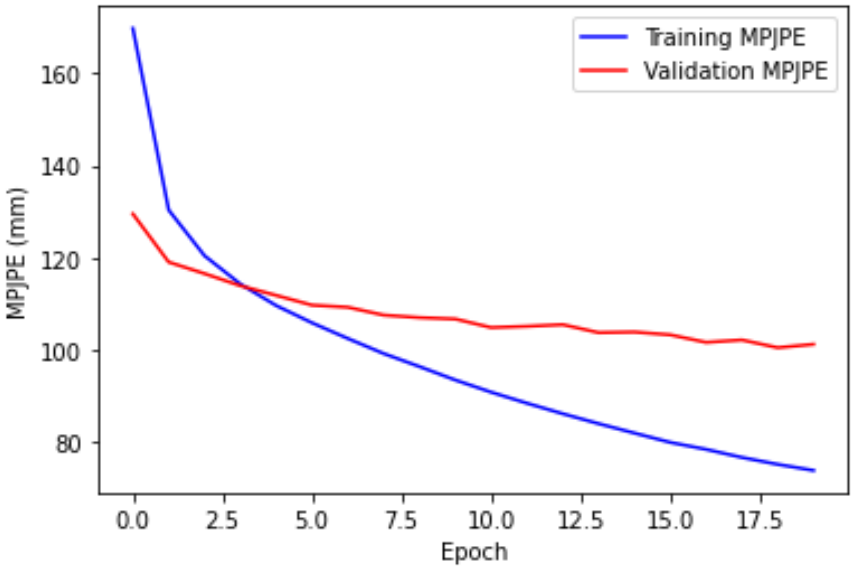

Figure 2: Training and Validation MPJPE as a Function of Epochs

The above plot very closely resembles the loss plot shown previously. Both the training and validation MPJPE values decreased quickly at first, and then decreased more gradually as more epochs were completed. Again we see the deviation between the training and validation curves, further proving that some significant overfitting began to take place. Additionally, just like the previous plot, we can see that these curves have not levelled off yet, indicating that training the model for more epochs may yield more accuracy. At the end of training, the final training MPJPE value was 73.95mm and the final validation MPJPE value was 101.27mm.

Conclusion

In conclusion, I was able to successfully build and train a CNN-LSTM-MLP model to identify 3D human joint positions in image sequences. The model's final MPJPE value was about 101mm, which is a good value considering the simplicity of the model. Unfortunately, I was severely limited by the physical system (mainly the small RAM and GPU memory amount) I used to build and train the model, and this meant that I had to use a model architecture that was complex enough to capture the necessary features but simple enough that I could successfully train it. Additionally, due to my small memory space, training the model took a very long time: about 16 hours or so to complete 20 epochs. Despite these limitations, I was able to train an accurate model, so I consider that a success!

Future Considerations

If I had more computing resources, there are a few changes I would make to my model to improve its performance. First, I would increase the model complexity so that it can more accurately capture features. For example, using an unmodified VGG-16 architecture or even a ResNet-18 architecture. Second, I would implement methods of combatting overfitting, such as using dropout or data augmentation, since the added model complexity would lead to significant overfitting (overfitting was seen in my simple model already!). To improve the model's accuracy even further, I would increase the batch size, add a learning rate decay, and I would run the model for more epochs.

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("DeepDynamicModel")`) 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 `evaluate.py`. This file can be used to evaluate my model's performance on a given set of testing data as follows:

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 the ground truth 3D positions for the 17 joints in each frame.
    - `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 hardware is not very powerful, so this was set to 3 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 and average MPJPE value.

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