Saaif Ahmed

661925946

ahmeds7

4/21/2022

# Introduction to Deep Learning: Programming Assignment #4 Report

## 1: Problem Statement

To develop a machine learning model that can accurately identify the location of joints in humans from a video of the requisite human. This problem can be posed as a dynamic regression model, where we take multiple frames of the video and condense then down between layers to receive the coordinates of the 17 joints that we are analyzing. The frames themselves are RGB found from the Human3.6M dataset. We build the model in a way where we can estimate the 3D body joint positions. Using Tensorflow we will build the model.

## 2: Model Description

The model is built as follows. Outside of the input layer there are 5 convolutional layers, 2 pooling layers, 2 fully connected layer, and an LSTM layer.

The 1st convolutional layer has 16 7x7 filters with no zero padding and stride of 2.

The 1st max pooling layer has an area of 2x2 with a stride of 1.

The 2nd convolutional layer has 16 3x3 filters with no zero padding and stride of 2.

The 3rd convolutional layer has 32 3x3 filters with no zero padding and stride of 1.

The 4th convolutional layer has 64 3x3 filters with no zero padding and stride of 1.

The 5th convolutional layer has 128 3x3 filters with no zero padding and stride of 1.

The 2nd max pooling layer has an area of 2x2 with a stride of 1.

The 1st LSTM layer has 128 nodes.

The 1st fully connected layer has 256 nodes.

The 2nd and final fully connected layer has 51 nodes

Each of the convolution layers, the LSTM layer, and the 1st fully connected layer has the ReLU activation functions which is defined as such.

$$ReLu(x) = \max\{0, x\}$$

The fully connected layer that follows from this serves as the output layer and has the linear activation.

$$Linear(x) = x$$

For each input we get an output $\hat{y}$ where for each of the 8 frames for each of the 17 joints we have 3 coordinates to determine where they are. As such the output layer is $\hat{y}^{8\times17\times3}$.

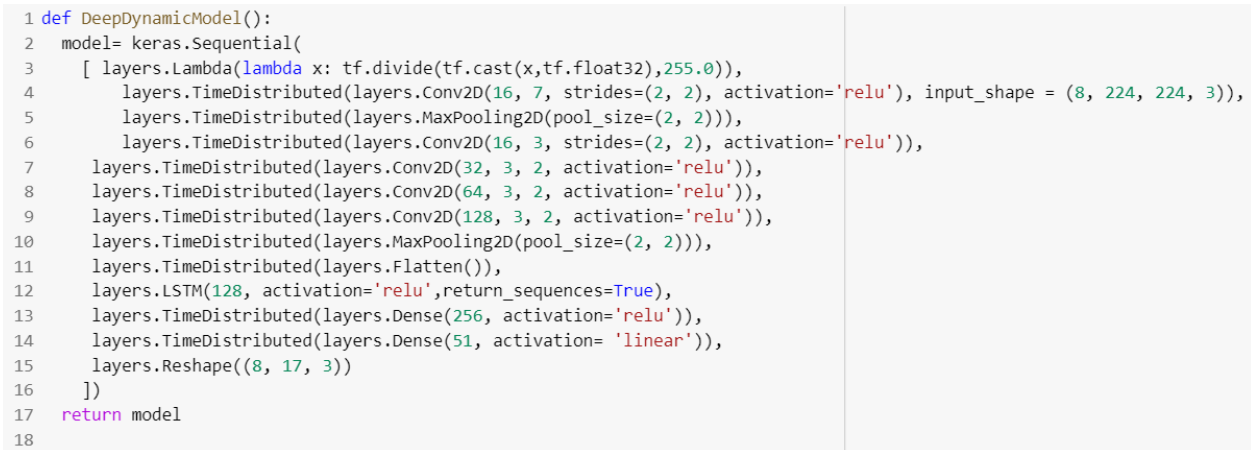
To optimize this model we minimize the mean-squared error loss function. For a ground truth data point $\vec{y}[m]$ and a predicted data point $\hat{y}[m]$ we define the function as

$$L(\vec{y}[m], \hat{y}[m]) = \sum_{m=1}^{M} (\vec{y}[m] - \hat{y}[m])^2$$

In order to further analyze the performance of the model we define the Mean Per Joint Position Error (MPJPE). This is given by the average Euclidean distance from the ground truth positions verse the predicting position for all data points.

$$\text{MPJPE}(\vec{y}[m], \hat{y}[m]) = \sum_{m=1}^{M} \sqrt{(\vec{y}[m][1] - \hat{y}[m][1])^2 + (\vec{y}[m][2] - \hat{y}[m][2])^2 + (\vec{y}[m][3] - \hat{y}[m][3])^2}$$

We construct the model in Tensorflow 2.X and as such the code definition to define the model is as follows.



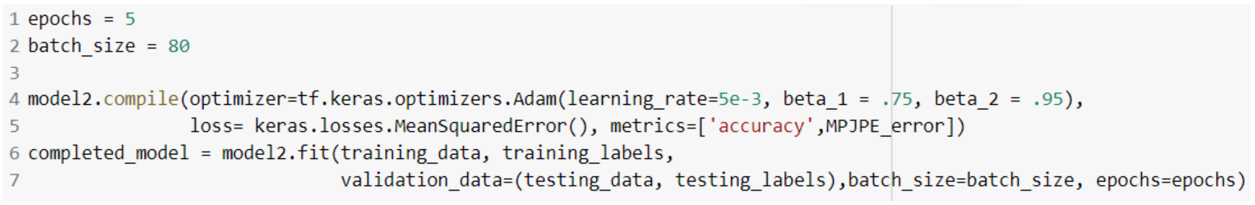
```python
def DeepDynamicModel():
  model= keras.Sequential(
    [ layers.Lambda(lambda x: tf.divide(tf.cast(x,tf.float32),255.0)),
        layers.TimeDistributed(layers.Conv2D(16, 7, strides=(2, 2), activation='relu'), input_shape = (8, 224, 224, 3)),
        layers.TimeDistributed(layers.MaxPooling2D(pool_size=(2, 2))),
        layers.TimeDistributed(layers.Conv2D(16, 3, strides=(2, 2), activation='relu')),
     layers.TimeDistributed(layers.Conv2D(32, 3, 2, activation='relu')),
     layers.TimeDistributed(layers.Conv2D(64, 3, 2, activation='relu')),
     layers.TimeDistributed(layers.Conv2D(128, 3, 2, activation='relu')),
     layers.TimeDistributed(layers.MaxPooling2D(pool_size=(2, 2))),
     layers.TimeDistributed(layers.Flatten()),
     layers.LSTM(128, activation='relu',return_sequences=True),
     layers.TimeDistributed(layers.Dense(256, activation='relu')),
     layers.TimeDistributed(layers.Dense(51, activation= 'linear')),
     layers.Reshape((8, 17, 3))
    ])
  return model

```

The model is built with tf.keras.layers and uses the respective library functions to build the model layer by layer.

***Note:*** *We normalize the data within the model itself we do not do it prior to passing it to the model. Please make sure that the testing data is not normalized by 255 when attempting to validate the model.*

To build and train the model we use the following code.



```python
epochs = 5
batch_size = 80

model2.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=5e-3, beta_1 = .75, beta_2 = .95),
                loss= keras.losses.MeanSquaredError(), metrics=['accuracy',MPJPE_error])
completed_model = model2.fit(training_data, training_labels,
                               validation_data=(testing_data, testing_labels),batch_size=batch_size, epochs=epochs)
```

The model is compiled use the Adam optimizer with the keras.losses.MeanSquaredError() loss as discussed above. We then use the model.fit() method to train and evaluate our model.

The model is saved using the model.save_weights() method and can be loaded with the model.load_weights() function in Tensorflow.

Something that will be discussed further in the discussion session is why the model was built this way. In short the dataset was extremely large and required lots of memory to load in. The model architecture, as a result had to be reduced in order to meet the memory limitations of the training process.

***Hyper Parameters:***

There are a few hyper parameters relevant to the model.

*Batch Size:* This determined the size of the mini batch runs per epoch of the model's training period. A small size took more time but was able to achieve convergence faster. For our model we used a batch size of 80.

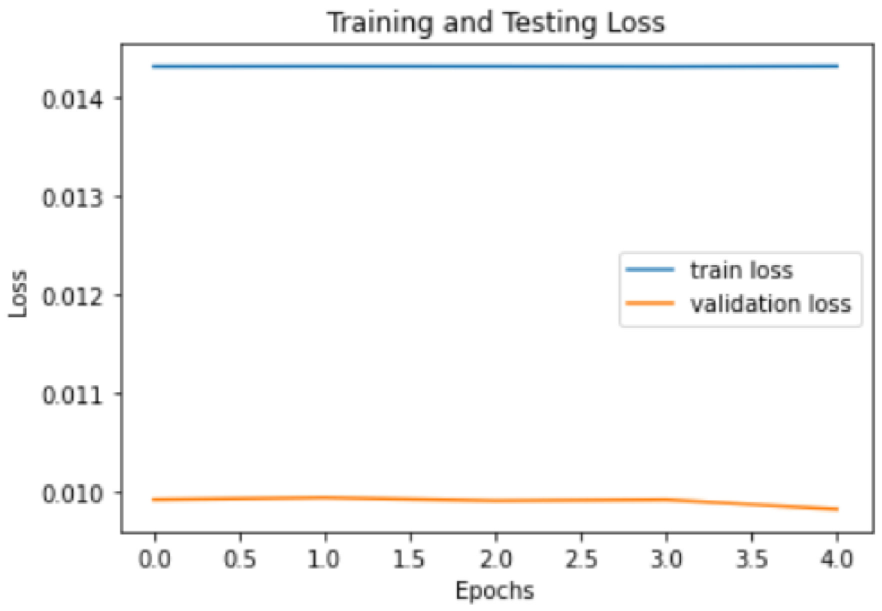
*Learning Rate:* This was a constant that was multiplied by the gradients to update the weights. We used a learning rate of 0.01

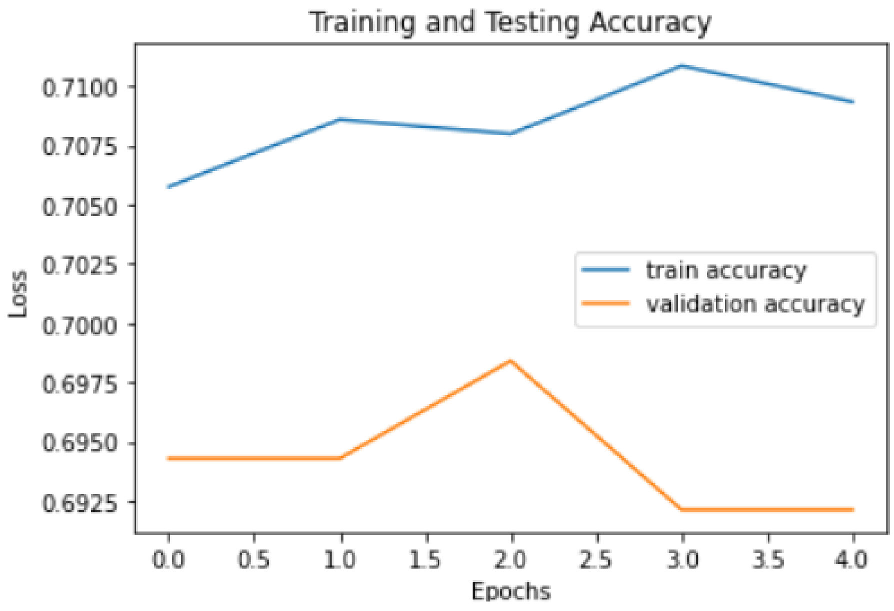
*Epochs:* This is synonymous with the iterations of the model's learning period. Because our network had many layers to it, 5 epochs were chosen to reach desired accuracy and convergence.

*Beta 1 & Beta 2:* The Adam optimizer in Tensorflow have two beta constants that determine exponential decay and growth rate of the model's learning. We diverge from the default setting by choosing Beta 1 as 0.75 and Beta 2 as 0.95.

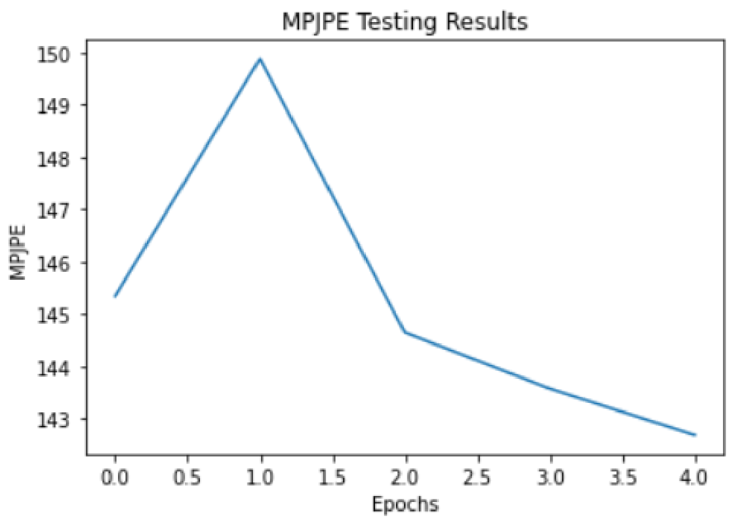
*Input Data:* The dataset used had 5,964 data points. As $D = \{\vec{x}[m], \vec{y}[m]\}, m = 1,2,..,M$ and input data $X^{5964\times8\times224\times224\times3}$ with output labels $\vec{y}^{8\times17\times3}$.

## 3: Results

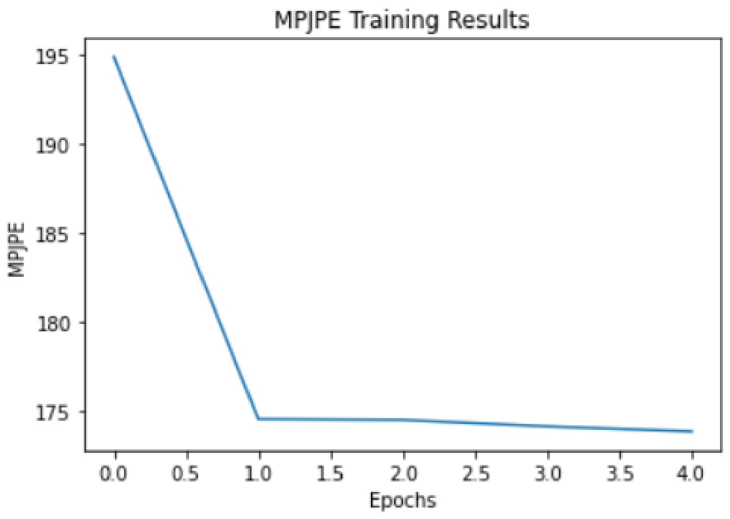




Above we show the Training Loss vs. Epochs, the Validation Loss vs. Epochs, the Training Accuracy vs. Epochs, and the Validation Accuracy vs. Epochs.



Above we show the MPJPE accuracy for all of the testing data points versus epochs of training. We reach the <150 mm margin quite comfortably and with further tuning of hyper parameters we can reach closer to the 100 mm margin. It is clear the error is minimized.



Above we show the MPJPE accuracy for all of the training data points versus epochs of training. We reach the <160 mm margin but it's only towards the end of training. With further tuning of hyper parameters we can reach closer to the 150 mm margin. Possibly increasing batch size would give a better representation.

```
MPJPE is: tf.Tensor(142.6551599891654, shape=(), dtype=float64)
```

Above we show the final MPJPE value. We see that the validation value has reached the <150 mm threshold as desired.

## 4: Discussion

To summarize the empirical results the model reached convergence after 5 epochs. Using Tensorflow's model.fit() function to train the model we reached >70% accuracy on the validation. The MPJPE performance did comfortably reach the <150 mm margin on the testing data set.

To discuss the runtime performance, training the model was very slow. This is due to the sheer size of the dataset being used. Over 8gb of data was used to train and test the model. Utilizing hardware accelerators such as a GPU would actually prevent us from training entirely because of system RAM limitations. This showcases another challenge with deep learning. Even with convolutional models to limit the size of the model sometimes a problem is simply too large. Even with 12gb RAM the model would crash if using the GPU due to the GPU requiring more RAM for Tensorflow operations. With 32gb of RAM potentially a larger model with larger batch sizes could have been used.

The usage of hyper parameters was useful in achieving desired accuracy levels. If using any learning rate on the $10^{-1}$ scale or higher, the model would hit a convergent point far below the desired accuracy, like 50%. Thus numbers of the $10^{-2}$ scale were used. The batch size was important as well. The accuracy of the model was dependent on how much data it could utilize to learn the weights in each epoch. Thus choosing a batch size that allowed for more learning per epoch led to better results. Adjusting the beta values in the Adam optimizer allowed the model to achieve that marginal difference in accuracy that made it reach the accuracy threshold. However pushing them below 0.80 prevented the model from learning at any desirable pace.

The Adam optimizer was the best optimizer possible within Tensorflow for this model. Other optimizers were tested but were unable to achieve reliable results with hyper parameter tuning. This gives rise to the reason of Adam being the most popular in industry. Node dropping was not implemented in the model definition. After analyzing the results of the data it is clear that some node dropping should have been implemented. The model was close to hitting the overfitting margin, but this was avoided because epochs were limited. Node dropping would have allowed us to achieve better representation from this.