Programming Assignment-4

Name: Mohaiminul Al Nahian

RIN: 662026703

Course No.: ECSE 6850

Introduction:

In this programming assignment, our goal is to implement a deep dynamic model to perform the task of human body pose estimation. The dynamic model consists of CNN layers to absorb the spatial information of data frames, LSTM layers to represent the temporal coherence between consecutive data frames and MLP layers to perform the final regression task for predicting 17 body joint co-ordinates. The dataset that we use is a subset of the Human3.6M dataset, where the input is video sequence of RGB frames and output is 17 body joint co-ordinates, each having a 3D value represented as a cartesian 3D point. The dataset that we have been provided has 5964 video sequence for training and 1368 video sequence for validation. Each video sequence consists of 8 frames and each frame has a dimension of 224x224x3 where the last dimension represents 3 RGB channels. The network we use has a pre-trained ResNet-50 architecture at the beginning, then a dense layer to downsample the number of connections, a flattening layer, then LSTM layer and finally output layer for regression. The network has been trained by minimizing the Mean Squared Error between the predicted output and the ground truth joint co-ordinate values. We tune our parameters by minimizing the loss function over several epochs by gradient descent approach. It is computationally inefficient to calculate gradients of all training samples at a time. So, We have implemented Stochastic Gradient Descent technique. The sizes of the layers were not fixed by the instruction of the assignment. So, through trial and error, we have chosen an architecture that makes a balance between the memory requirement of the machine that we use for training and the final performance. we have tuned the hyper-parameters of the model through trial and error as well. In each epoch, we train over all mini-batches and keep track of training & testing loss and training & testing Mean Per Joint Position Error (MPJPE). After finishing the training, we check the model's performance on the validation data and save the learned parameters in the format specified.

The rest of the report is organized as follows: We describe the architecture of the model, then we discuss about the loss function and hyper-parameter settings, we plot the results of MPJPE and loss over epochs, we do a result analysis and finally we close the report with a discussion. We have also included the code used to train the model.

Model Description:

The architecture used is this dynamic model consisting of a CNNs, LSTMs and MLPs. We pass the input which is a video sequence of 8 frames into a pretrained ResNet-50 architecture trained on the imagenet dataset. Each input frame has a size of 224x224x3. The input values are normalized by dividing the pixel values by 255. We discard the final classification layer of ResNet and take features from the previous layer which gives a map size of (S, 8, 7, 7, 2048). Here S denotes the size of the minibatch and 8 denotes the time distributed 8 consecutive frames. In order to reduce computation overhead, we use a dense layer before passing it to the LSTM. So, we get a total feature map size of (S, 8, 7, 7, 128). We flatten this layer to vectorize it and get a map size of (S, 8, 6272). We pass this layer to the LSTM layer having 256 units.

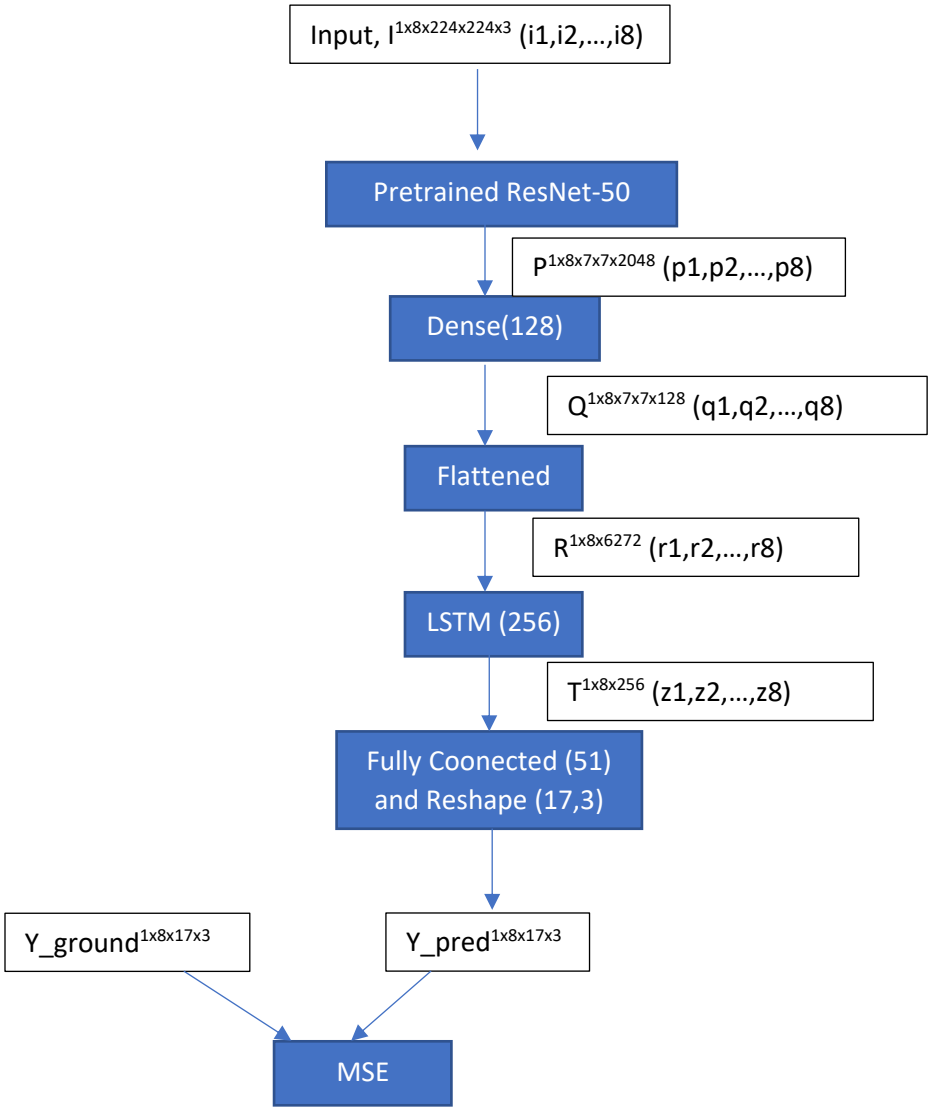

Figure: Dynamic Deep Model Architecture

The output of the LSTM (S, 8, 256) is fed into the fully connected output layer consisting of 51 nodes which gives an output of (S, 8, 51). Finally, we reshape the output to (S, 8, 17, 3) where the final 2 dimensions represents the 17 joints' cartesian co-ordinates.

It should be mentioned that the activation function used in all the intermediate layers, apart from the ResNet layer is ReLu. But the output layer has a linear activation, meaning that we consider the output layer values without giving any activation. A common mistake people make is use softmax activation in the output layer. A visualization of the model architecture is given here. Here the structure is shown for 1 data points or in other words, a batch size of 1, where the number of consecutive frames is 8.

Loss Function:

This is a regression problem. So, the loss function chosen to solve this problem is the mean squared error. Mathematically Mean Square Error is defined as

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\left(y_{true} - y_{pred}\right)^2$$

Now, the output values in this assignment are 3D joint co-ordinates of 17 joints per frame. The output layer has 51 nodes corresponding to the x,y,z co-ordinates of 17 joints per frame. So, if we consider the ground truth and predicted output shape as 51 node vectors, corresponding to 17 3D joints, we can write the MSE loss per frame as

$$loss = \frac{1}{51}\sum_{k=1}^{51}\left(y_{true}[k] - y_{pred}[k]\right)^2$$

Now, each data point has 8 frames, so we must add the loss of the 8 consecutive frames to get the total loss of 1 frame. If we have a batch size of S , then the total loss over one iteration would be

$$Loss = \sum_{i=1}^{S}\sum_{j=1}^{8} loss_{ij}$$

Or, $$Loss = \frac{1}{51}\sum_{i=1}^{S}\sum_{j=1}^{8}\sum_{k=1}^{51}\left(y_{ij,true}[k] - y_{ij,pred}[k]\right)^2$$

We minimize this loss over each iteration by gradient descent approach.

Optimizer:

To optimize the loss function, we have used Adam Optimizer. Adam is an optimization algorithm for stochastic gradient descent for training deep learning models. Adam combines the best properties of the AdaGrad and RMSProp and can handle sparse gradients on noisy problems. The hyper-parameter setting description of the Adam optimizer is given in a following section.

Hyper-parameter Setting:

The learning rate setting was a challenging task for this assignment. Too big a learning rate makes training loss zigzag over epochs. Too small values were making the training process very slow. After multiple trial and error, we have used a learning rate for the Adam optimizer to be 0.0005 with beta1=0.9 and beta2=0.999 and all other values to be default. We have also carefully chosen a batch size of 4 after playing with different batch sizes and running the code for several epochs. A bigger batch size was causing the code to get an Out of Memory Error. For this particular assignment, we have not used any special regularizing parameters such as dropout or batch normalization to keep the model more simple, because even without any regularization parameter, the trained network was showing good results on the validation data.

Plots of Loss and MPJPE vs Epochs:

The training has been done for a total of 20 epochs. After that, the test loss no longer decreases, rather starts an upward trend, indicating overfitting. The loss and MPJPE curves are given below:

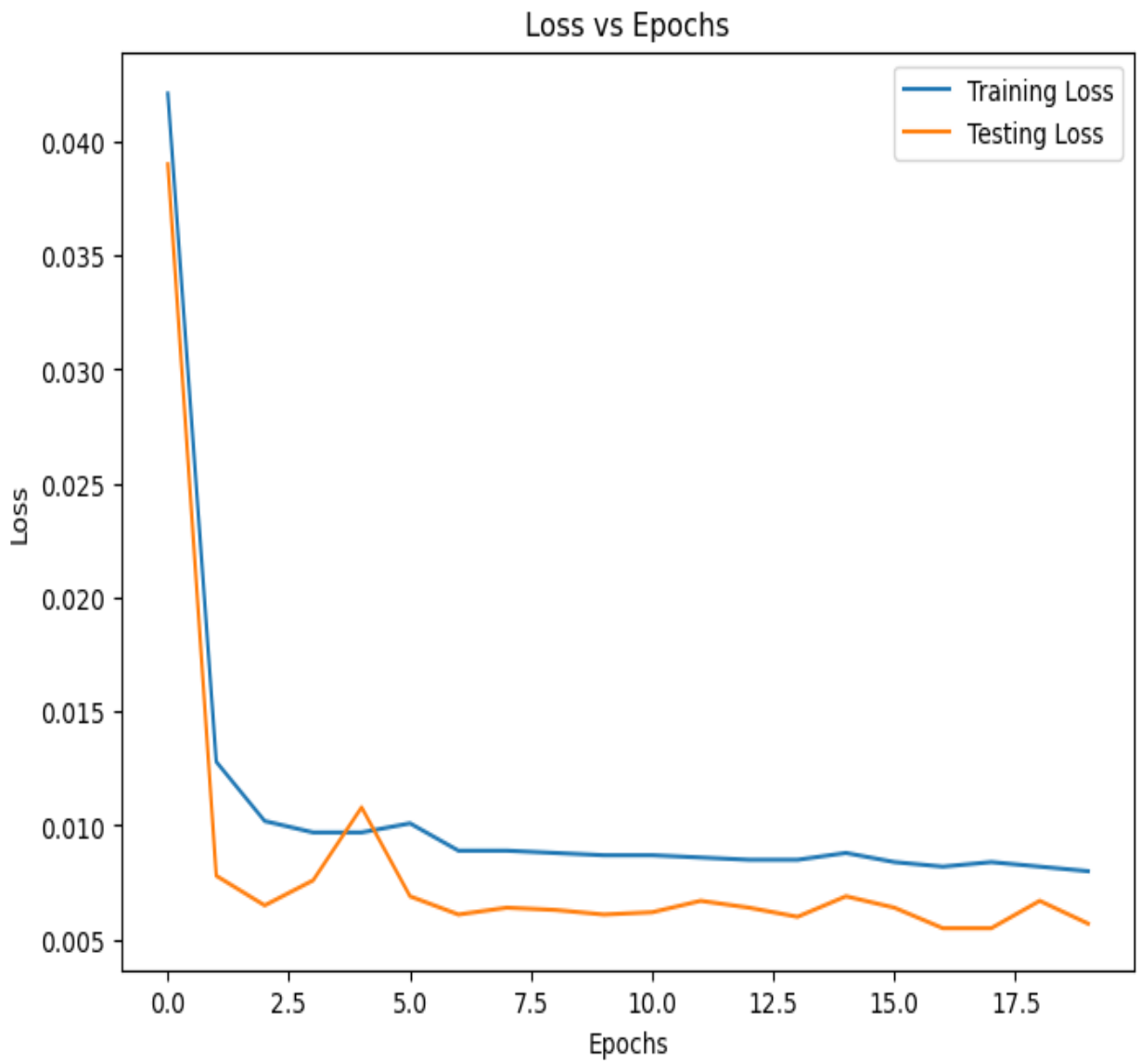

Figure: Train and Test Loss vs Epoch

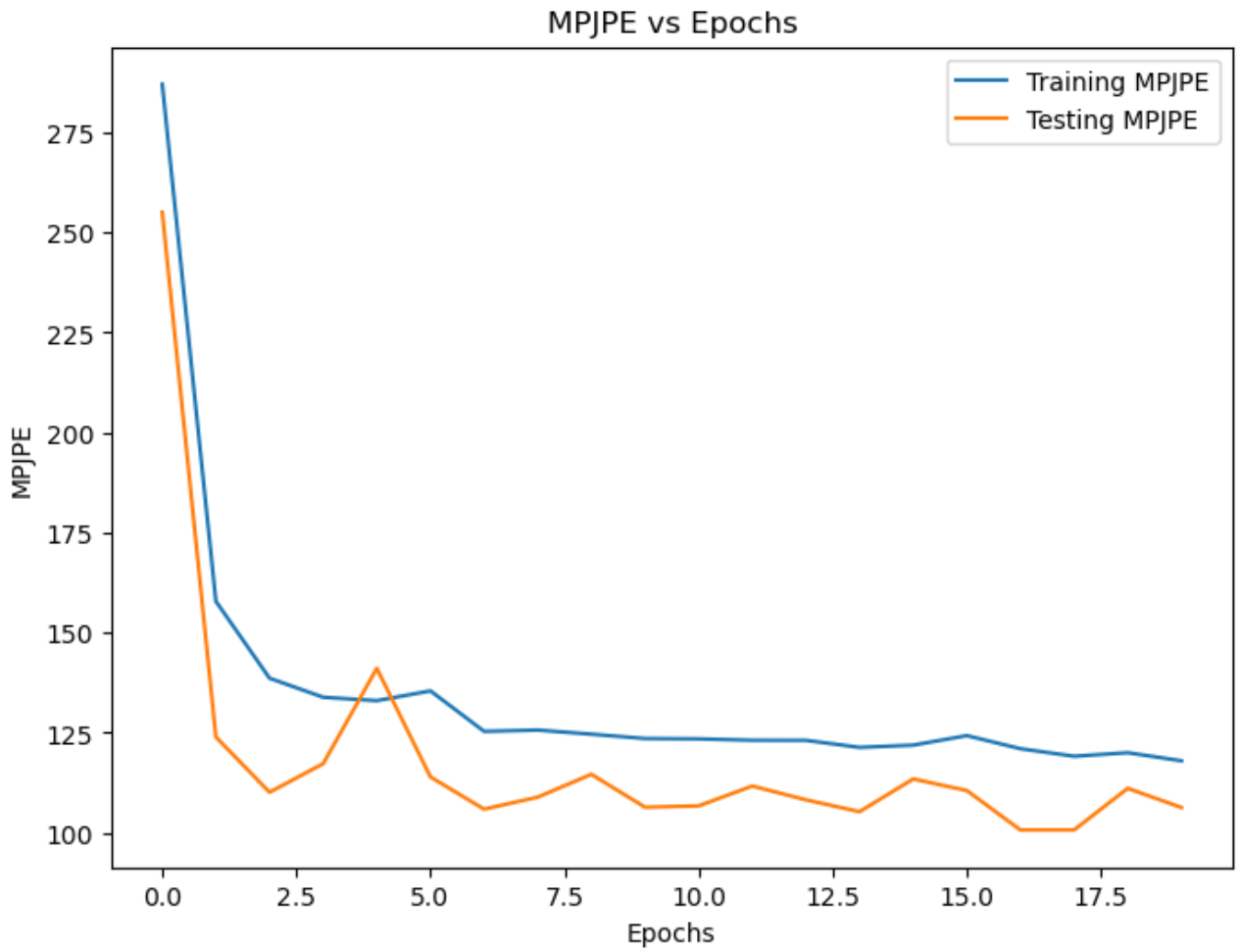

Figure: Train and Test MPJPE over Epochs

Result Analysis (Mean Per Joint Position Error):

The performance metric used in this experiment is the Mean Per Joint Position Error (MPJPE) of the validation dataset which is the average Euclidean distance between predicted 3D positions and the ground truth positions on 17 joints in 8 frames for all test data points. Using the saved model weight, we calculate the MPJPE on the entire test dataset. The learned model yields an **MPJPE of 114.6535 on the validation dataset.**

Discussions:

Of all the programming assignments in this course, this one was the most challenging in terms of making a balance between computational resource constraints and model performance. The dataset was large npy files which already took large portion of the

memory resources. Then we had to define a model that can handle the computational overhead without being out of memory. I used a pretrained ResNet-50 model trained on imagenet dataset at the front end. According to the example instruction, we are supposed to connect the output of a ResNet to the LSTM layer directly. However, as ResNet-50 is a fairly big architecture and the feature that I took from it had a size of 8x7x7x2048, so I down sampled it by a dense layer to 8x7x7x128 before vectorizing and passing it to LSTM layer. Otherwise, I was getting out of memory error when trying to connect ResNet with LSTM directly. Also, the LSTM layer is computationally heavy, so I had to downsize the total number of units here from 1024 to 256. Even 512 units raised an OOM error here. The batch size was also tweaked at the same time. Starting with a batch size of 16, I had to downsize the batch size to fit the computational resource constraints. At last, I could manage a batch size of as high as 4 only. Smaller batch size was making the training loss go up and down over epochs. The learning rate choosing had to go through some tweaking as well. I used 0.001, 0.005, 0.0005, 0.0001 as learning rates. With bigger learning rates, the loss reduced rapidly but the performance got stuck and was not improving over epochs. On the other hand, the smallest learning rate was making the training process very slow. So, eventually, I settled for a learning rate of 0.0005. With all these trial and errors, the model could achieve an MPJPE of 114.6535mm on validation data which is fairly low, considering the minimum requirement of 150mm. It should be mentioned that, while training for longer periods of time, the model was starting to overfit, i.e, the validation performance was degrading. This could be because I have not used any dropout or other regularizers in this experiment and considering the size of the model, the available training data was not that big, so the model was starting to memorize the training data. So, I had to stop training at a good point where the training and validation performance was fairly high. Overall, implementing different knowledge gained from the course lectures we could get a moderately good performance from this deep dynamic network for human body pose estimation from the Human3.6M dataset.

Appendix: Code For Training:

```
# %%

import matplotlib.pyplot as plt

import numpy as np

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import models,initializers,regularizers

from tensorflow.keras.layers import Reshape, Activation, Conv2D, MaxPooling2D, Flatten, Dense, LSTM,
TimeDistributed, Lambda, Dropout, BatchNormalization, GlobalAveragePooling2D
```

```
from tensorflow.keras.applications.resnet50 import ResNet50
# %%
# Loading training data

x_train = np.load('../input/program4data/data_prog4Spring22/videoframes_clips_train.npy', mmap_mode='c')
# Loading test data
x_test = np.load('../input/program4data/data_prog4Spring22/videoframes_clips_valid.npy', mmap_mode='c')
# Loading train labels
y_train = np.load('../input/program4data/data_prog4Spring22/joint_3d_clips_train.npy', mmap_mode='c')
# Loading test labels
y_test = np.load('../input/program4data/data_prog4Spring22/joint_3d_clips_valid.npy', mmap_mode='c')
print('Data Loaded Successfully')
# %%
print('x_train shape:', x_train.shape)
print('x_test shape:', x_test.shape)
print('y_train shape:', y_train.shape)
print('y_test shape:', y_test.shape)
# %%
# Model Hyperparameters
minibatch_size = 4
epoch = 20
learn_rate = 0.0005
# Computing mpjpe
def mpjpe(y_true, y_pred):
    val = tf.math.multiply(1000.0,tf.reduce_mean(tf.norm(y_pred-y_true, ord='euclidean', axis=3)))
    return val

# Defining a Model
model = models.Sequential ([
    Lambda(lambda x: tf.divide(tf.cast(x,tf.float32),255.0)),
    TimeDistributed(ResNet50(input_shape=(224,224,3),include_top=False, weights='imagenet')),
```

```
    TimeDistributed(Dense(128)),
    TimeDistributed(Flatten()),
    LSTM(units=256, return_sequences=True),

    TimeDistributed(Dense(51, activation='linear')),
    Reshape((y_train.shape[1],y_train.shape[2],y_train.shape[3]))

])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learn_rate), loss='mean_squared_error',
metrics=['mse',mpjpe])
dependencies = {
    'mpjpe': mpjpe
}

history = model.fit(x_train, y_train, epochs=epoch, batch_size = minibatch_size, verbose=1,
validation_data=(x_test, y_test))

# %%
model.summary()

# %%
# Saving the Model

model.save_weights("model_weights")



# Saving Loss and MPJPE by epochs

np.save('trainhistory.npy',history.history)
```

```
# %%
# Loading Loss and mpjpe by epochs

all_history=np.load('trainhistory.npy',allow_pickle='TRUE').item()

train_loss = all_history["loss"]
test_loss = all_history["val_loss"]
train_mpjpe = all_history["mpjpe"]
test_mpjpe = all_history["val_mpjpe"]

# Train Loss by Epochs

plt.plot(train_loss, label ="Train Loss")
plt.plot(test_loss, label ="Test Loss")
plt.legend()
plt.xlabel('Total number of epochs')
plt.ylabel('Loss')
plt.show()

# mpjpe by Epochs

plt.plot(train_mpjpe, label ="Train mpjpe")
plt.plot(test_mpjpe, label ="Test mpjpe")
plt.legend()
plt.xlabel('Total number of epochs')
plt.ylabel('mpjpe')
plt.show()



# %%
#Model Performance on Test Data
```

```
minibatch_size = 4
test_dataset = (
    tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(minibatch_size)
)


test_dataset = (
    test_dataset.map(lambda x, y:
                     (tf.divide(tf.cast(x, tf.float32), 255.0),tf.cast(y, tf.float32))))
tmp_MPJPE = []
tmp_loss = []
for x, y in test_dataset:
    predict = model.predict(x)


    loss = tf.math.reduce_sum(tf.losses.mean_squared_error(y, predict)) / minibatch_size


    MPJPE = tf.math.reduce_mean(tf.math.reduce_euclidean_norm((y - predict), axis=3)) * 1000
    tmp_MPJPE.append(MPJPE)
    tmp_loss.append(loss)
testing_los = tf.reduce_mean(tmp_loss)
test_MPJPE = tf.reduce_mean(tmp_MPJPE)
print('MPJPE:', test_MPJPE.numpy())
print('Avg Test Loss:', testing_los.numpy())
```