PA 4: Human Pose Estimation with Recurrent Neural Networks

1 Introduction

In this programming assignment, you will train a recurrent neural network for the task of human pose estimation on the YouTube pose dataset. Specifically, we will analyze video data from YouTube videos and estimate the joint positions of human upper body. These videos are sequences of images in certain resolution, and will be treated as the sequential input to our recurrent neural network(RNN). However, the RNN cannot directly take a sequence of images as the input, so we will first apply a convolutional neural network(CNN) to learn some features from the original images, and then feed these learned features to the RNN. The following guide will take you through the curated version of this human pose estimation dataset and help you create a model in TensorFlow.

1.1 Human Pose Estimation

The problem of human pose estimation, defined as the problem of localization of human joints, has enjoyed substantial attention in the computer vision community. We can formulate the pose estimation problem as a regression problem. More specifically, we predict the joint positions(i.e. the x, and y coordinates in the image coordinate system) given the input images. In general, there are totally 19 well-defined joints, e.g., head, right hip, and right knee, left wrist, and so on. In order to simplify this task, we are going to focus on 7 upper body joints, i.e. the head, right shoulder, left shoulder, right wrist, left wrist, right elbow, and finally the left elbow. We are going to estimate the position of these joints.

2 Dataset

For the purpose of this programming assignment, some preprocessing steps are applied to curate the dataset into a convenient format to use. The dataset for this assignment can be downloaded here. You are also encouraged to look at the original data, but DO NOT use it for this assignment.

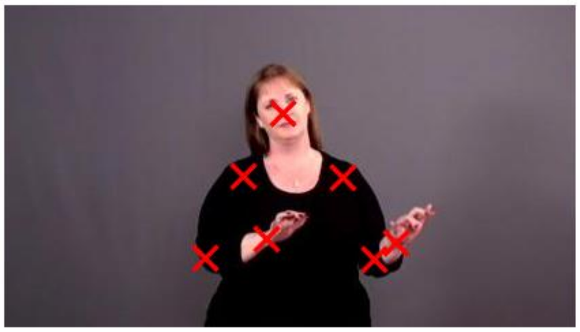
2.1 Original Dataset

The videos in the original dataset were downloaded from YouTube and automatically scaled and cropped (using an upper body detector) so that the shoulder width is around 100 pixels wide. 100 frames from each of the scaled and cropped videos were randomly chosen and manually annotated with upper body joints. The Head, Right wrist, Left wrist, Right elbow, Left elbow, Right shoulder, and Left shoulder were annotated. The Ground Truth(GT) for a frame is a 7 by 2 matrix containing 2D locations for the upper body parts, column 1 are x values and column 2 are y values. Rows are formatted from top to bottom as: the Head, right wrist, left wrist, right elbow, left elbow, right shoulder, and the left shoulder.

2.2 Preprocessing

For simplicity and also for the purpose of evaluation, I did all the preprocessing steps for you, so that the data we are using is consistent. The key steps of the preprocessing are briefly discussed as the following.

- First, the resolution for different YouTube videos varies. Typically, the videos are around 400×600, I scaled all 50 videos into 64×64 small videos for you to use. Also, I adjust the GT labels according to the scale factor applied to the original videos accordingly. So that the curated labels are correctly located on the upper body joints. The following images are the original image and the scaled image and also the corresponding labels plotted on each image.



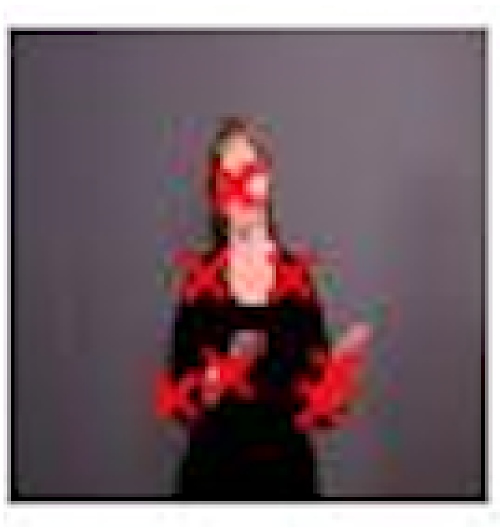


Figure 1: The original image with labels(top) and the scaled image with labels(down)

- Second, we need to create sequences from these frames, the strategy is also pretty straightforward, we select consecutive frames of length 10 in the same videos with overlaps, i.e. the first sequence starts from frame 1 to frame 10, and the second frame starts from frame 2 to frame 11, and we keep doing this until we reach the end of a video, and then we do the same operations to other videos, and so on. Then, we select every other consecutive frames in the same videos also with overlaps, i.e. the first sequence starts from frame 1, then frame 3, frame 5 up to frame 19, the second sequence starts from frame 2, then frame 4, up to frame 20, this will give us additional sequences. Consequently, we end up with a dataset of totally 8600 sequences of images, for each sequence, we have 10 images, and each image is a 64 by 64 by 3 RGB image. I will give you the first 7000 sequences for training, and the rest will be used to evaluate your model, so you will need to decide, for the 7000 training data, how much data you should use for training and how much for validation.

2.3 Load the dataset

First you should download and unzip the dataset, the size of the compressed file is 700MB, so make sure you have a good internet connection. Then load the data using pickle in the following way,

```
import pickle
# load the dataset into memory
data_file = open('youtube_train_data.pkl', 'rb')
train_data, train_labels = pickle.load(data_file)
data_file.close()
```

the resulting *train data* is a (7000, 10, 64, 64, 3) ndarray with dtype np.int8, and the *train labels* is a (7000, 10, 7, 2) ndarray with dtype np.float32. From the data preprocessing section, you should be able to interpret the shape here without surprise. With a 4GB Memory or more, you should be able to load and convert all the data to np.float32.

2.4 Data normalization

For this assignment, you will also need to apply similar normalization techniques as the previous assignments, you will need to normalize the original images by subtracting the mean and dividing the standard deviation. Note that the mean and standard deviation should be computed over each image or each sequence, since the images here are much more diverse then the previous Cifar10 dataset, computing the mean across different videos may not work well. There are many tricks for normalization, and the method we are using may not the best one, but for the purpose of evaluation, I will normalize the withheld dataset in the same manner, so that I can test your model. You may normalize the data like the following.

```
# normalize the training data over each image
train_data -= np.mean(train_data, axis=(2, 3, 4), keepdims=True)
train_data /= np.std(train_data, axis=(2, 3, 4), keepdims=True)
```

3 Model

We will build up a RNN with Long-Short Term Memory(LSTM) cells in this assignment. Recall that RNN is a special kind of Neural network, where the output of the cell from last time step together with the input of the current time step will be taken as the input of the current cell, the compact and unrolled structures are shown as the following, the figure here can also be found in this post.

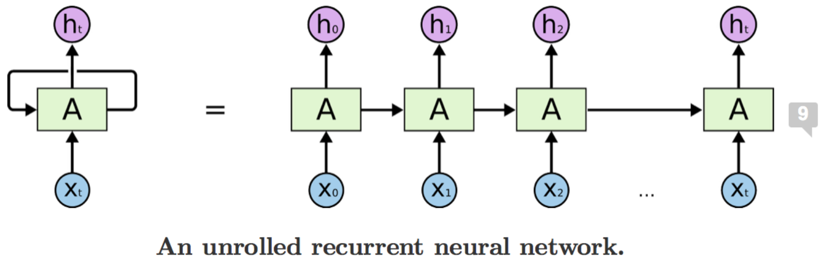


Figure 2: General RNN architecture

In Figure 2, X_t is the input vector of the t^{th} time step, and the h_t is the output of the t^{th} time step, so the h_t together with x_{t+1} will be fed into the cell in the $t+1^{th}$ time step. In our case, we will have 10 cells, since the sequence length is 10. The number of hidden nodes is left to you as a hyper parameter you can tune. The input for each time step is a RGB image which cannot be directly fed into an LSTM cell, so a CNN will be stacked before feeding images into the LSTM cell to learn some features from the images, that is the X_t in the above Figure 2 is the last fully connected layer of a CNN applied on the t^{th} image of a sequence.

3.1 Model Design

In order to avoid overfitting, images in a sequence can share the same CNN, i.e. the CNN takes 10 images from a sequence as the input, and output 10 vectors, each of the 10 vectors is an input for each time step of the RNN. A reference architecture of this model is shown in Figure 3. You are also welcome to design your own CNN model, for example, you can use 3 convolution layers, apply the RelU activation function to add some nonlinearities, and then followed by 3 max pooling layers, one for each activation layer, then flatten the final feature map to form the fully connected layer. When feeding the image sequence, apply this CNN to each of the 10 images, and get 10 feature maps, these feature maps can be flattened and fed in to the corresponding LSTM cells of the RNN. After we collect the output of all 10 LSTM cells, for each output H_t, you will use some weights to map it to the final output layer Y_t, the dimension of the Y_t should be a vector with 14 elements, i.e. we have 7 joints, and for each joint, we have 2 coordinates (x, y). Note that the weights here can also be shared among different LSTM cells and their corresponding outputs. Since this is a regression problem, no activation function is needed at this output layer, and you can use the mean squared loss to train your model.

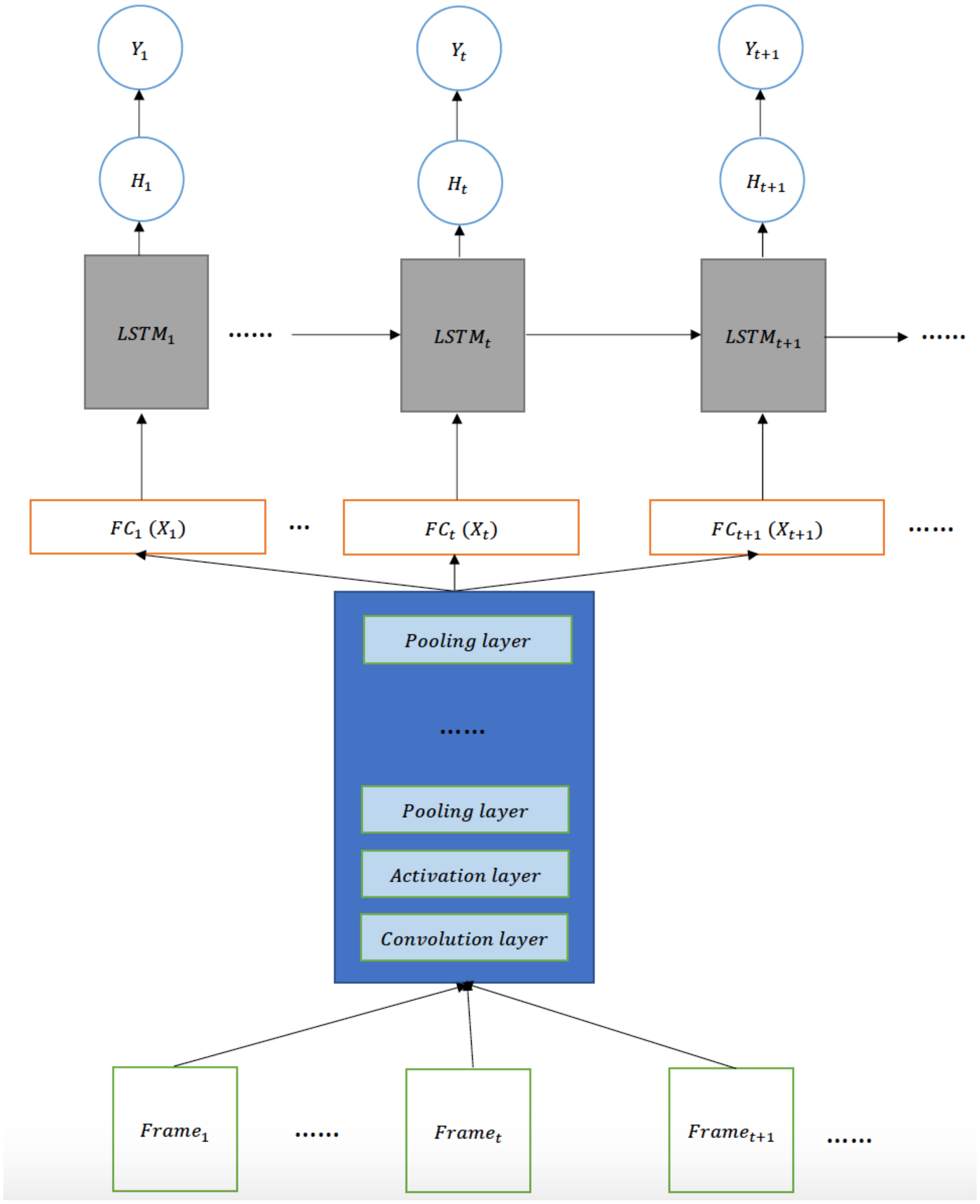


Figure 3: A reference model architecture: FC_t refers to the fully connected layer for the t^{th} input frame, H_t refers to the output of the t^{th} LSTM cell, and Y_t is the prediction for the t^{th} input which should be a vector with 14 elements representing 7 joint positions

4 Implementation and key functions

4.1 CNN

The CNN part of the model is trivial as you already finished the last assignment. Similar functions and initialization techniques can be applied here. Keep in mind that you cannot directly feed the feature map into the RNN, so the output of your CNN should be the last fully connected layer which is a vector.

4.2 RNN

First, you will need to define an LSTM cell with certain number of hidden nodes(num_units) with the function tf.nn.rnn_cell.LSTMCell. Then take the output of this function and the output of your CNN(the shape of the output of your CNN should be $[batch\_size, 10, num\_nodes\_fc\_layer]$) as the input of the function tf.nn.dynamic_rnn and produce the output for all 10 cells, i.e. the h_val in the following functions. The shape of the output h_val should be $[batch\_size, 10, num\_units]$

```
# instantiate a LSTM cell
lstm_cell = tf.nn.rnn_cell.LSTMCell(num_units)
# define your RNN network, the length of the sequence will be automatically retrieved
h_val, _ = tf.nn.dynamic_rnn(lstm_cell, rnn_input, dtype=tf.float32)
```

Once you have the output for all the LSTM cells, you can produce the final output and compute the mean squared loss. For the final output, you may compute it by a for loop if the weights are shared by different cells, the shape of the output layer should be [batch_size,10, 7, 2].

```
# collection of all the final output
final_output = tf.zeros(shape=[batch_size, 0, 7, 2])
for i in np.arange(seq_length):
    temp = tf.reshape(h_val[:, i, :], [batch_size, num_units])
    output = tf.matmul(temp, w_fc) + b_fc
    output = tf.reshape(output, [-1, 1, 7, 2])
    final_output = tf.concat([final_output, output], axis=1)
```

PLEASE NOTE: You may compute the final output without using for loop as well. We have shown you just one approach. In fact, if you can do it without a for loop, it will speed up the computation.

4.3 Training

Once you define the loss function, TensorFlow will take care of everything for training, typically the AdamOptimizer is a good option with a appropriate learning rate. The idea here is that it keeps track of the history of the gradients, and use the history to calculate the next step size for the gradient descent. Feel free to vary the batch size also, I would recommend you start from a small batch size, like 5, since this net is relatively complicated compared with the previous CNN, so a large batch will result in a longer computing time for the gradient.

During training, you will need to plot the training loss and testing loss every certain iterations. While training, you may also plot the predicted joint positions on the images to check whether the prediction makes sense or not. A sample plot for the prediction may look like the following Figure 4. The final average testing error over different joints should be around 10 pixels or less.

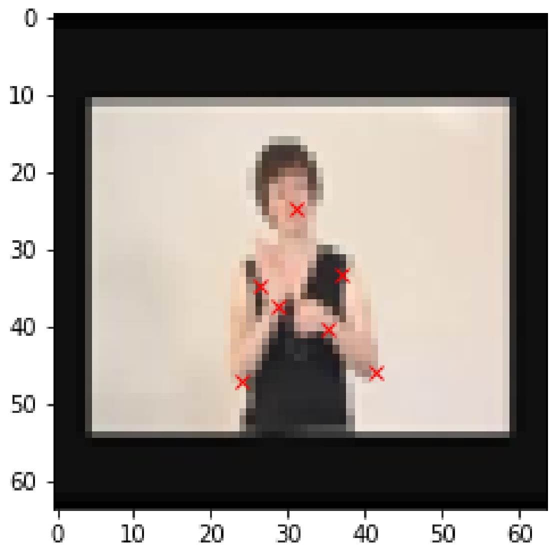


Figure 4: Image and the predicted joint positions

In addition to including the final average pixel distance error on the validation dataset you select(at least 100 sequences should be used to validate this error). You will also need to calculate and plot the accuracy curve for all 7 joints. Here the accuracy is defined as the percentage of image for a predicted joint position that is within certain pixels as the GT position. You will need plot 7 curves, one for each joint, the curves may look like this.

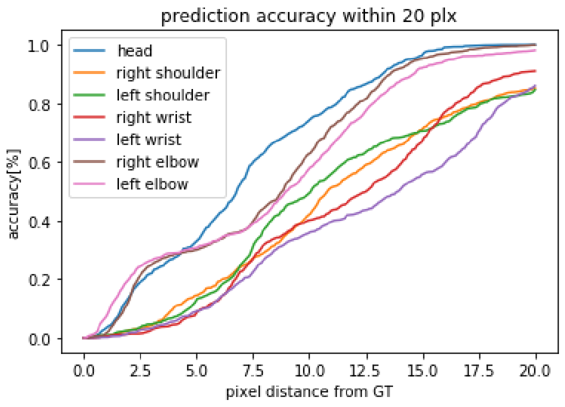


Figure 5: Prediction accuracy

For instance, the blue curve in Figure 5 shows the accuracy for the head, that is, about 60% of the predicted position is within 7.5 pixel distance error to the GT, and all of the predicted position is within 20 pixel distance error to the GT. Pay attention here that the distance is not your mean squared loss, it is the distance between the GT and the predicted position.

5 Saving your Model

IMPORTANT! FOLLOW THESE INSTRUCTIONS CAREFULLY.

In order for us to validate your model, it has to be in a certain format. In this regard, we have to follow certain steps.

First, you are supposed to use Tensorflow version 1. The following are few ways you can use Tensorflow version 1:

1. *In your local machine*: Import Tensorflow using the following command:

   ```
   import tensorflow.compat.v1 as tf
   tf.disable_v2_behavior()
   ```

   More information about this can be found in https://www.tensorflow.org/guide/migrate

2. *Colab:* You may choose to Colab which is a free online platform by Google for machine learning purposes which also provides free GPUs. If you use Colab, use the following command to import Tensorflow version 1:

   ```
   %tensorflow_version 1.x
   import tensorflow as tf
   ```

Second, before starting to define your tensorflow graph, please include the following line in your code:

```
tf.reset_default_graph()
```

This resets the global default graph and hence ensuring that any nodes from your previous tensorflow graph definition is not included in your final submission. *From our side, we require it prevent certain hassles related with naming of the tensors*.

Third, your tensorflow graph is supposed to have an input and an output:

1. Input: It is a batch of video frames. Each batch consists of 10 frames and each frame is an unprocessed RGB image of size 64*64 whose pixels are of type **uint8**. This is same as the video frames in youtube_train_data.pkl. Any pre-processing has to be a part of the tensorflow graph. This is because Deep Learning models are supposed to be *end-to-end models*. The definition of the input in your tensorflow graph MUST be:

   ```
   input_frames = tf.placeholder(dtype=tf.uint8, shape=[None, 10, 64, 64, 3], name='input_frames')
   ```

   In the above code, **None** helps us feed batches of any size. I repeat, you MUST NOT change the above command in any way.

2. Output: It is a tensor of shape [batch_size, 10, 7, 2] and of type **float32**. Its $(i, j, k, 0)^{th}$ element and $(i, j, k, 1)^{th}$ element are the x and y coordinates respectively of the k^{th} joint of the j^{th} frame of the i^{th} batch. The order of the joints should be same as that in youtube_train_data.pkl file. This output tensor MUST be named **joint_pos**. Please note that you MUST name your tensor using the command **name = 'joint_pos'**. For example, if you have used the code in Section 4.2 as your final output layer, then you can simply use *tf.identity()* function to copy **final_output** as **joint_pos** and also name it. This can be done as follows:

```
joint_pos = tf.identity(final_output, name='joint_pos')
```

Fourth, your tensorflow graph should not require any other input but **input_frames** to compute **joint_pos**. To ensure this, the following command must run without error:

```
predictions = sess.run(joint_pos, feed_dict = {input_frames: data})
```

where, **session** is the name of the tensorflow session and **data** are the video frames of the test dataset.

Fifth, we need access to the tensors **input_frames** and **joint_pos** of your TensorFlow graph to evaluate your model. In order to facilitate this, we ask that you use a Graph Collection . These nodes should be stored in a collection called "validation nodes". The following code demonstrates how to do that.

```
input_frames = tf.placeholder(dtype=tf.uint8, shape=[None, 10, 64, 64, 3],
name='input_frames')
.
.
.
 joint_pos = ….
.
.
.
# Create the collection.
tf.get_collection("validation_nodes")
# Add stuff to the collection.
tf.add_to_collection("validation_nodes",input_frames)
tf.add_to_collection("validation_nodes",joint_pos)
```

Please Note: The order you run the **tf.add_to_collection**() should also be the same as the above code.

Sixth, when you are done training, you will need to save your model in a form that allows us to grade it. For this, we ask that you use the built-in TensorFlow Saver class. You can save your model using the following code:

```
# start training
saver  =  tf.train.Saver()

with  tf.Session()  as  session:
      .
      . # Train your model
      .
     # this saver.save() should be within the same tf.Session() after the training is done
     save_path  =  saver.save(session,  "my_model")
```

After you are done with the training the above code should generate four files which MAY look something like:

checkpoint

my_model.data-00000-of-00001

my_model.index

my_model.meta

PLEASE NOTE: We will be making an announcement in LMS with a link to a Python script which you can use to verify if these four files are generated as per our requirements.

6 Submission

You need to submit the following:

1) Compress the report and the python code as a single zip file and submit through LMS. The details about what should be included in the report is available in the grading policy (which will be uploaded soon).
2) The four files: checkpoint, my_model.data-00000-of-00001, my_model.index, my_model.meta should be submitted via the google drive link which we already emailed you. Please do not zip these four file before uploading to google drive. Also, please do not put these files in a sub-folder. Directly drop these files in the google drive link.