program4_report

April 24, 2020

1 Introduction

In this programing assignment4, we are instructed to implement a recurrent neural network model and apply the model to estimate the gesture pose in the youtube video.

The data contains 7000 samples. Each sample contains ten RGB images (64 x 64 x 3), each corresponds to one frame. We are also instructed to split the data into the training set and the validation set. I randomly choose 6000 of them as the training data and the rest of them as validation data.

Due to the input is an array of time-dependent images, we need to build appropriate representation for our Neural Network. A neural network is like a brain, which can process the bioelectrical signal and relay on sensors, like eyes, to translate the visual signal to the appropriate inputs. The best input for the neural network is tensor, so we need to build the sensors for our neural network, which is convolutional modules. Apart from the sensor that connects the NN with the outside world, the NN also requires a “sensor” that takes within system signals as input, here we choose the recurrent module.

2 Theory

2.1 CNN

Simply put, the purpose of introducing a CNN module is to have the network learned appropriate filters that can translate the pixels signal to a tensor.

This tensor should have several dimensions. Here, I choose 128, which means that the following NN of CNN should simultaneously consider these 128 features as the input.

An intuitive example in this assignment is like the CNN takes in an image and outputs 128 features, like, finger, mouth, ..., so our NN can do reasoning base upon these features.

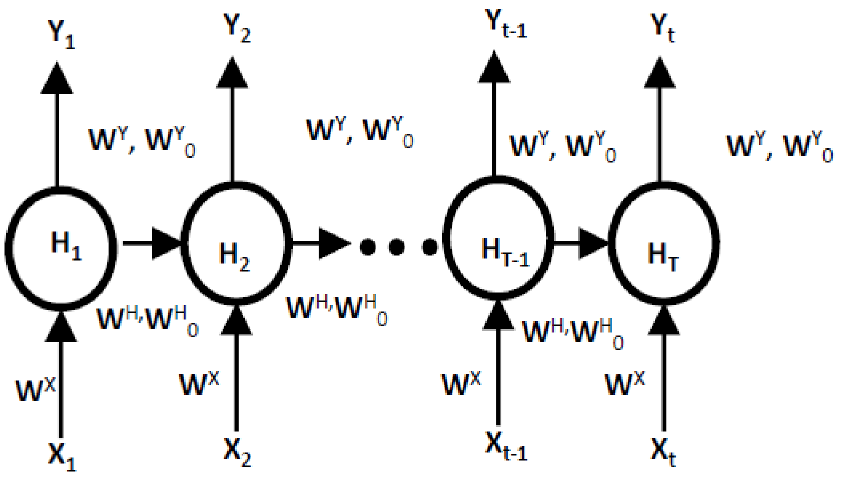
2.2 RNN

2.2.1 Purpose of RNN

The purpose of using a recurrent neural netowrk (RNN) is **to capture the sequential dependency between data points.**

2.2.2 Basic RNN structure

Figure 1 (from lecture note-chapter 4-page16) shows a basic RNN structures.



- X_t is the input at time t. In this assignment, the input is a vector or a feature map extracted from a CNN.
- H_t is the hidden state at time t. It is the memory of the network, and the input information for the next time step, telling the time dependency information
- Y_t is the output of the cell at time t. It is also regarded as the output of the model if the cell t is the last step of the time sequence.
- W^x, W_0^x, W^h, W_0^h are the input weights for each cell. More specifically, they are the weights for the rectified linear function where the activation function in this module is usually tanh. For exmaple: $H_t = \tanh(W^x X_t + W_0^x)$. Note that all the t in this time sequence shared these four weights.
- W^y, W_0^y are the output weight. Just like any output weight of the nerual network, W^y, W_0^y is usually follow by a softmax or sigmoid function. W^y, W_0^y are also shared by all the samples within the same sequence.

2.2.3 General Back-propagation through time

Compared with BP, BPTT is different in 1) the gradient is time-dependent 2) all gradient should be averaged over time-domain before used to update the weights.

To deal with the time-dependen gradient, we need to first figure out the relationship between each node. The whole structure of the RNN is built in a directed Markov way. The nodes at the head of an arrow only depend on the nodes at the arrow tails. Hence, the backpropagation is to merely backprop the gradient from the arrowhead nodes to the arrow tail nodes. So we need to initiate from the cell at time step T, compute the gradient nodes by nodes until $t = 1$.

$$W_t^y, W_{0,T}^y \rightarrow W_t^h, W_{0,T}^h \rightarrow ... \rightarrow W_1^x, W_{1,0}^x$$

Calculating the gradients of Y_t, $W_t^y, W_{0,t}^y$, $W_t^h, W_{0,t}^h$ and $W_t^x, W_{0,t}^x$, are simple becomes it only depends on the gradients of their outputs. For example:

$$\bigtriangledown W_t^y = \frac{\partial Y_t}{\partial W_t^y} \bigtriangledown Y_t$$

and

$$\bigtriangledown W_t^x = \frac{\partial H_t}{\partial W_t^x} \bigtriangledown H_t$$

The gradient of H_t are slightly more complicated, becasue its has two output. What we need to do is to add both gradient together.

$$\bigtriangledown H_t = \frac{\partial H_{t+1}}{\partial H^t} \bigtriangledown H_{t+1} + \frac{\partial Y_t}{\partial H^t} \bigtriangledown Y_t$$

The second problem is to update the weights, we need to average over the gradients across time domain, for example:

$$\bigtriangledown W^y = \sum_{t=1}^{T} W_t^Y$$

We are allowed to do this because the weights are shared by all the cells in this time sequence.

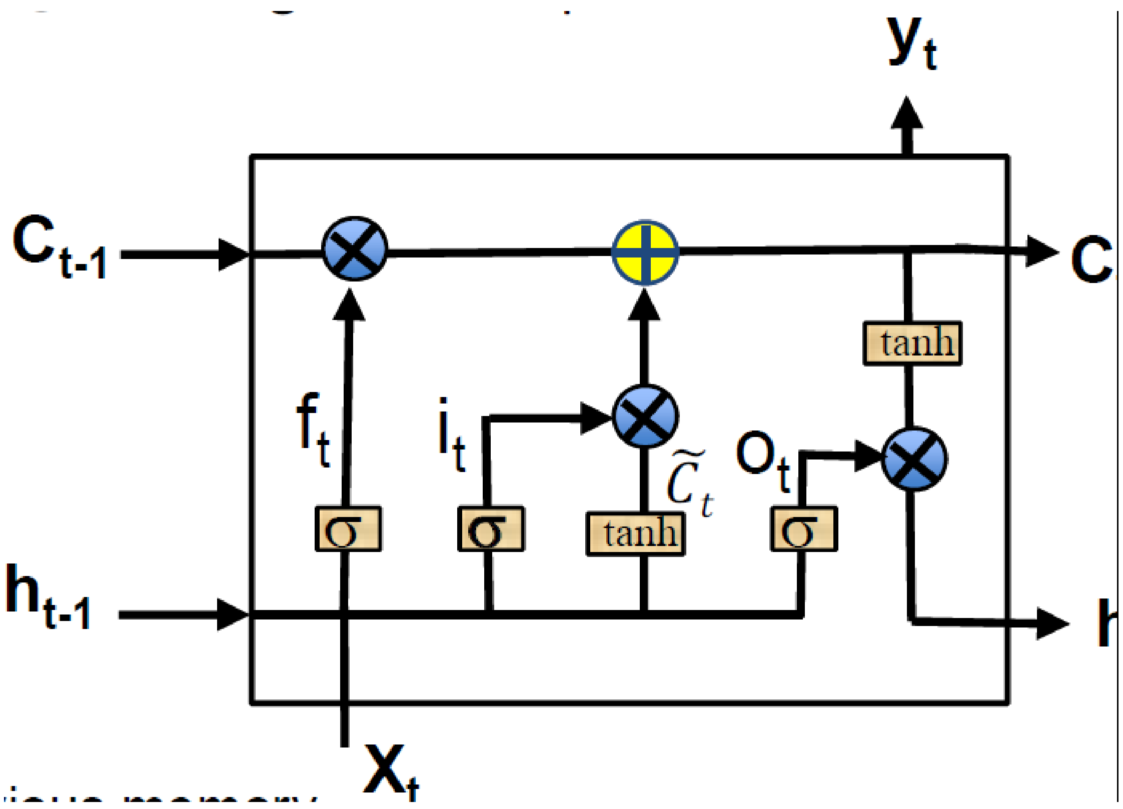
As the result of BPTT, the gradient of the early cell is the multiplication of a pile of following gradients. This can be dangerous because once some gradients are smaller than 1 (even not small), the multiplication of them will return 0, which is called gradient vanishing. Similarly, if the multiplication of those large gradients will cause gradient explosion. The solution is we need to build in some mechanism to constraint the range of the gradient.

This why we need Long Short Term Memory (LSTM).

2.3 LSTM

LSTM is to design a special memory cells, imposing a limit to the range of gradient value cleverly. The core idea is using gate structure.

Figure2 (Lecture note, Chapter4, page 30) shows the structure of one LSTM cell.



The equation of this cell are:

- $i_t = \sigma(W^{h,i}h_{t-1} + W^{x,i}x_t + W_0^{x,i})$
- $f_t = \sigma(W^{h,f}h_{t-1} + W^{x,f}x_t + W_0^{x,f})$
- $o_t = \sigma(W^{h,o}h_{t-1} + W^{x,o}x_t + W_0^{x,o})$
- $\tilde{C}_t = \tanh(W^{h,c}h_{t-1} + W^{x,c}x_t + W_0^{x,c})$
- $C_t = C_{t-1} \odot f_t \oplus \tilde{C}_t \odot i_t$
- $h_t = \tanh(C_t) \odot o_t$

We can see LSTM basically introduces a bi-memory system. One of these two memories h_t guide the current jobs while C_t is a more high-level memory, which does not directly participate in the current computation.

These gates constraints the range of the gradient. However, I am not able to use equations to show how they realize this because this is not explicitly covered in our lecture.

2.4 Model architecture and experimental setting.

2.4.1 Introduction of the data

The input data contains 7000 samples. Each sample contains ten RGB images (64 x 64 x 3), each corresponds to one frame. We are also instructed to split the data into the training set and the validation set. I randomly choose 6000 of them as the training data and the rest of them as validation data. Basically, it is short clip of the video.

What we want the NN model do is to output a (7 x 2) matrix for each frame. The 7 here represents the number of the joint and 2 indicates the space

The target we are training the NN towards is the ground truth label of the joints and we want our NN model can approach the performance of the ground truth.

2.4.2 Architecture

The following diagram shows the architecture I used:

$$\begin{align}
&\text{Input}(32 \times 32 \times 3) \rightarrow \tag{1}\\
&\text{Conv Module} \rightarrow \text{vector}(128) \rightarrow \tag{2}\\
&\text{RNN Module} \rightarrow \text{LSTM nodes}(32) \rightarrow \tag{3}\\
&\text{output}(7 \times 2) \tag{4}
\end{align}$$

I used the previous CNN architecture:

$$\begin{align}
&\text{Input}(32 \times 32 \times 3) \rightarrow \tag{5}\\
&\text{Conv1}(5 \times 5 \times 32) \rightarrow \text{ReLU} \rightarrow \text{Maxpool}(2 \times 2) \rightarrow \tag{6}\\
&\text{Conv2}(5 \times 5 \times 32) \rightarrow \text{ReLU} \rightarrow \text{Maxpool}(2 \times 2) \rightarrow \tag{7}\\
&\text{Conv3}(3 \times 3 \times 64) \rightarrow \text{ReLU} \rightarrow \text{Maxpool}(2 \times 2) \rightarrow \tag{8}\\
&\text{FC}([11 * 11 * 64] \times 128) \tag{9}
\end{align}$$

2.4.3 Loss function and Measurement

To minimize the discrepancy of the model prediction and the ground turth, we need to first define a loss function and a metric to quantify the distance.

To define the loss fuction I used is the euclidean distance before the predictive space coordinate indices:

$$err = \sqrt{(\hat{x}_i - x_i)^2 + (\hat{y}_i - y_i)^2}$$

where $\hat{x}_i$ is predictive value, and the one with no hat is the ground turth.

The metric to quantify the performance is basically the same, except that we are instructed to turn this continuous error into a logical term by asking:

$$err < n \text{ pxiels}$$

If the distance error is smaller than n pixels, we return 1, otherwise 0. The range of n is [1,20].

2.5 Hyperparameter

There are many hyperparameters, I will list all of them. But to tell the truth, I haven't tested all of them. I also switch my learning rate and batch_size during the training, so I will only make some discussion in these two subsections.

2.5.1 Optimizer and Learning rate:

I use Adam as recommended by TA and only try the learning rate 0.001 and 0.002. The latter one significantly shorten the training iteration, while does not hurt the performance.

2.5.2 Batch size

This has been a trivial problem for me all the time until I trained this RNN. In this RNN, if we want to do batch training, we need to reshape the tensor frequently. The reshaped tensor usually has batch_size x sequence in the first dimension, which can exceed the GPU memory. For me, I first chose 500 as batch size but my computer does not work because 500 x 10 is equivalent to 5000 batch size. After I resize it to 100, everything is good.

2.5.3 Initializer

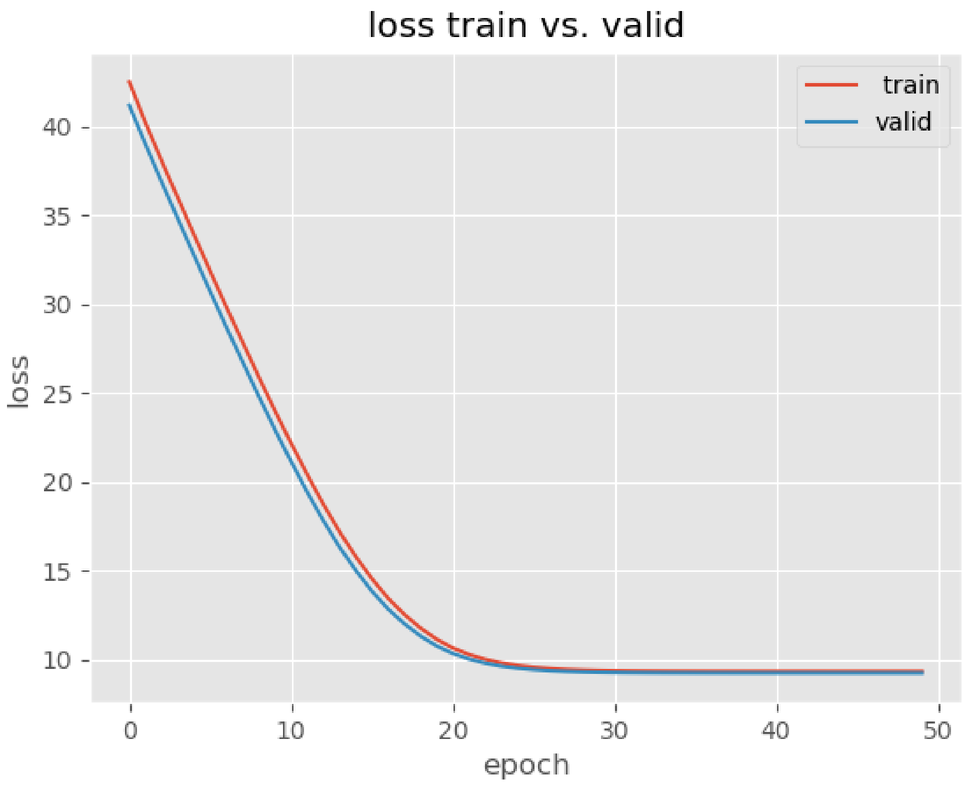
I use Xavier to initialize all weights, and constant 0 biases.

2.5.4 Features size and RNN output size

These are also hyperparameters, but I just picked a number arbitrarily. My observation is that computer vision people prefer those 2 to the power of n. So I pick 128 and 32. My psychology background told me, 128 is too much for visual information, 64 might be more consistent with the psychology theories.

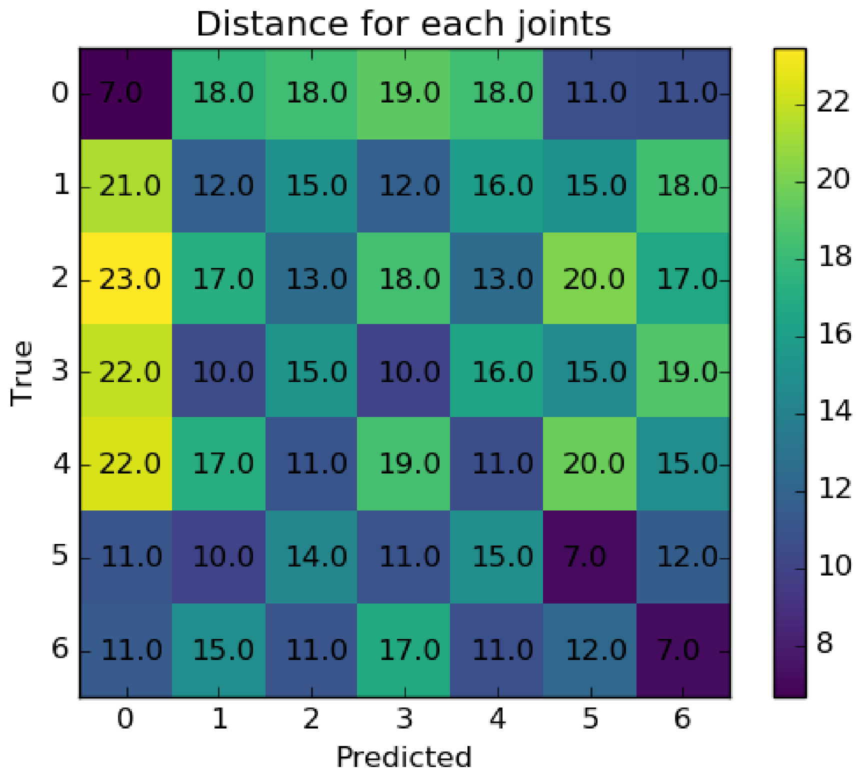
3 Results

3.1 Overall loss: training and validation



This is the figure for the overal loss of both training and validation set. I plot the loss vs. the epoch so it looks pretty smooth. The loss converge at around 27 epoch. The optimal loss stablize at arround 9.2.

3.2 Average pxiel distance.

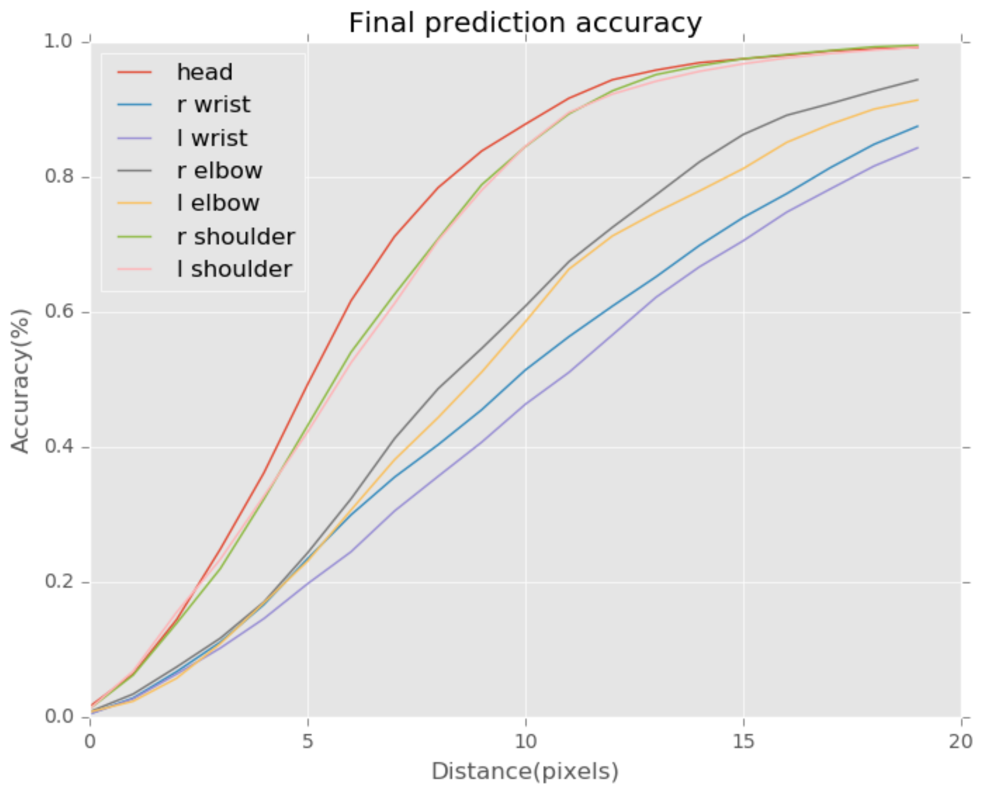


Here I plot something very similar to the confusion matrix. The darker means, the smaller the distance is, and light means a large distance.

We can focus on the diagonal line if we only care about the performance of the model. The predictions of 0: head 5,6: two shoulders are better than others. The predictions of other joints are slightly worse than these three, maybe because elbows and wrists are too subtle.

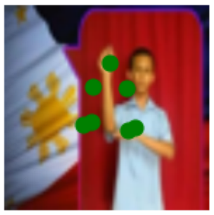
Look at other pixels, we can try to hypothesis the inadequacies of the model. For example, 2 & 4 and 1 & 3 are close to each other, which is acceptable between these joints are adjacent. The problem their distances to other joints are still similar, which means the prediction might be at the same location. Here I guess maybe the model does not do well in extracting the features and is confused by the space location. If I want to make any improvement, working on the convolutional module can be prioritized.

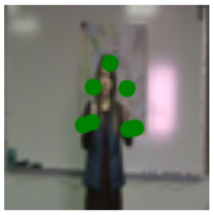
3.3 Cummulative distribution function



This figure shows the quality of the predictions for each joint. We can see the head and shoulders can be predicted most accurately, which also matches our intuitions: they are obvious and takes more pixels.

3.4 Random samples





From these samples, we can see the head and shoulders predictions are OK and It is hard to distinguish which prediction is wrist and which one is elbow.