Introduction to Deep Learning Final Project

Jiawei Xu

May 5, 2019

1 Overview

In this programming assignment, we were asked to design and train an RNN model for action recognition on the data set UCF-11. The data set contains sequences of actions under 11 different categories. The instructors have processed the data so that each sequence contains 30 frames of an action, and each frame has a size of $(64, 64, 3)$. Therefore, every mini-batch has a size of $(batch\_size, 30, 64, 64, 3)$. After being loaded, the data were normalized and shuffled since the data provided were well ordered by their categories. The labels were converted to follow the 1-of-k encoding convention. I tried 3 different models for this task. One of them worked well so far. The first model was a 3D convolution neural network developed based on a related article[1]; the second one was a VGG neural network; the third was a variation of the one we used for the programming assignment 4[2].

2 Structures

For this project, I tried to use different structures mentioned in the document, but ended up having bad performances from all of them. I tried to do the grid search on the hyper-parameters and tried to use optimizers relatively insensitive to hyper-parameters like Adam optimizer. Before the discussion on the performances, I would like to introduce the in-depth of the structures I used.

2.1 C3D

Similar to 2D convolutions, 3D convolutions take in a batch of data and return the predictions for each data in the batch. However, in order to deal with the temporal information in the sequences of frames, the 3D convolution filters convolve not only through widths and heights, but also through frames. For example, the $(3, 3, 3)$ filter collects data from 3 frames in a sequence with the size of $(3, 3)$. The structure of the 3D convolution neural network I used for the project is shown in Figure 1.

There are a total number of six 3D convolution filters and four max-pooling filters. All filters have a small kernel size of $(3, 3, 3)$ since the data given to us have a small resolution. All max-pooling layers have a size of $(2, 2, 2)$ and a stride of 2 except for the first one which has a size of $(1, 2, 2)$ because I want to capture the information of all frames in a sequence in the early convolution stage. The related paper suggested the last dense layers to have 4096 nodes, but limited by the computational capability of my computer, I chose to make them to contain 1024 nodes each. The last fully connected layer will then generate the prediction vector with softmax activation. This model did not work well.

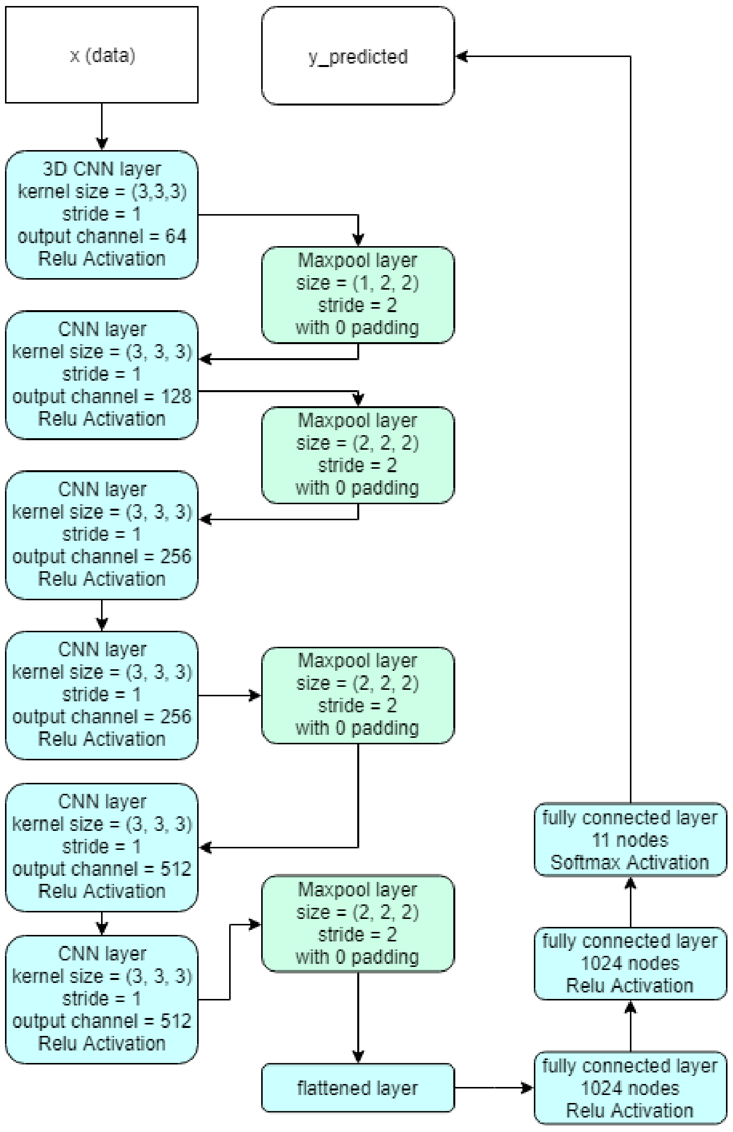
x (data)
y_predicted
3D CNN layer kernel size = (3,3,3) stride = 1 output channel = 64 Relu Activation
Maxpool layer size = (1, 2, 2) stride = 2 with 0 padding
CNN layer kernel size = (3, 3, 3) stride = 1 output channel = 128 Relu Activation
Maxpool layer size = (2, 2, 2) stride = 2 with 0 padding
CNN layer kernel size = (3, 3, 3) stride = 1 output channel = 256 Relu Activation
CNN layer kernel size = (3, 3, 3) stride = 1 output channel = 256 Relu Activation
Maxpool layer size = (2, 2, 2) stride = 2 with 0 padding
CNN layer kernel size = (3, 3, 3) stride = 1 output channel = 512 Relu Activation
CNN layer kernel size = (3, 3, 3) stride = 1 output channel = 512 Relu Activation
Maxpool layer size = (2, 2, 2) stride = 2 with 0 padding
flattened layer
fully connected layer 1024 nodes Relu Activation
fully connected layer 1024 nodes Relu Activation
fully connected layer 11 nodes Softmax Activation

Figure 1: C3D Structure

2.2 VGG9

VGG16 is a convolution neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition"[3]. The model has its advantage in classifications of images. However, since all the convolution actions are 2D, the model does not have the knowledge on the temporal data. Due to the small scale of our data set, I modified the model so that it only has 9 filters instead of 16. The structure of the model is shown in Figure 2.

The CNN takes in each frame of a sequence and generates the classification result of 500 classes. All convolution filters have a kernel size of $(3,3)$ and a stride of 1. All max pooling layers have a size of $(2,2)$ and a stride of 2. The number of channels go from 3 to 64 and is multiplied by a factor of 2 each time a max pooling is done. After the convolutions, the resulted tensor is flattened and connected to two fully-connected layers of 2048 nodes, and then the output layer of 500 nodes. Then the results from the frames in a batch of sequences are stacked together to form a tensor of size $(batch\_size, len\_sequence, classes)$. The tensor is flattened while keeping the dimension of the batch to form a matrix of size $(batch\_size, len\_sequence \times classes)$. The result then goes through a linear conversion (multiplication and addition) to generate the final classification result. This model did not work well, obviously, because it did not learn, or learned really a little from the last linear operation layer outside the CNN, on the temporal information in the sequences.

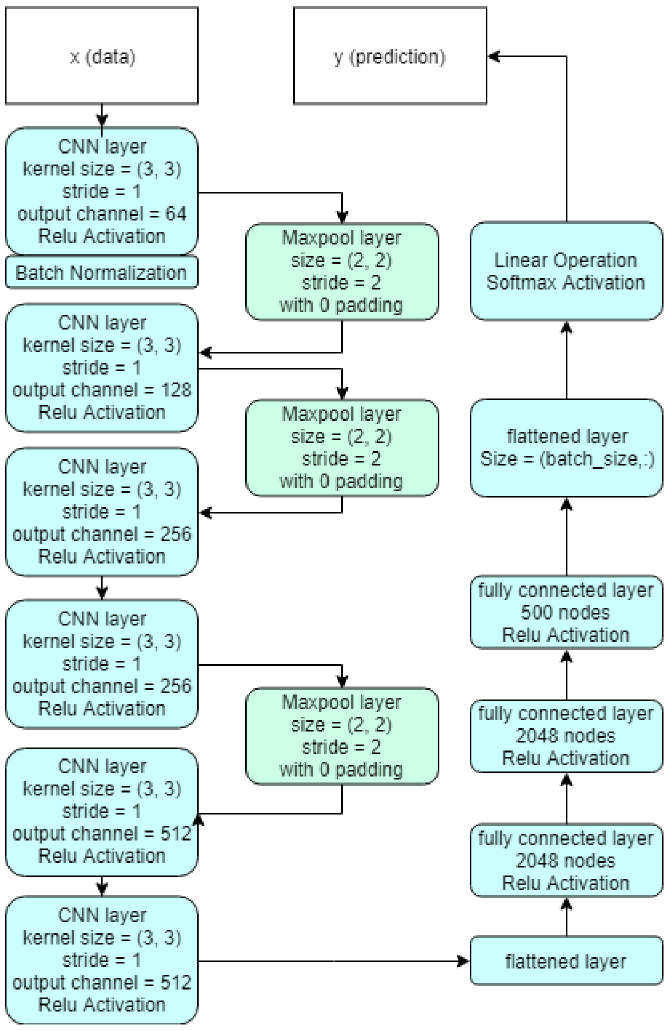
x (data)
y (prediction)
CNN layer kernel size = (3, 3) stride = 1 output channel = 64 Relu Activation
Batch Normalization
Maxpool layer size = (2, 2) stride = 2 with 0 padding
Linear Operation Softmax Activation
CNN layer kernel size = (3, 3) stride = 1 output channel = 128 Relu Activation
Maxpool layer size = (2, 2) stride = 2 with 0 padding
flattened layer Size = (batch_size,:)
CNN layer kernel size = (3, 3) stride = 1 output channel = 256 Relu Activation
fully connected layer 500 nodes Relu Activation
CNN layer kernel size = (3, 3) stride = 1 output channel = 256 Relu Activation
Maxpool layer size = (2, 2) stride = 2 with 0 padding
fully connected layer 2048 nodes Relu Activation
CNN layer kernel size = (3, 3) stride = 1 output channel = 512 Relu Activation
fully connected layer 2048 nodes Relu Activation
CNN layer kernel size = (3, 3) stride = 1 output channel = 512 Relu Activation
flattened layer

Figure 2: VGG9 Structure

2.3 LSTM

LSTM, as was discussed in programming assignment 4, stands for long-short term memory. As an RNN cell, it captures the temporal data in a sequence without having the trouble dealing with the gradient vanish or gradient explosion. Similar to what we have done in PA4, the batch of sequences are fed into a CNN to generate preliminary labels. Then the labels are stacked up together to feed into the LSTM cells. The structure of the model is shown in Figure 3.

There are three convolution layers and three max=pooling layers in the CNN. The first convolution layer has a small window size of $(2, 2)$, allowing the neural net to capture as much information as possible at the early stage of the convolution. The second has a window size of $(3, 3)$, and the third has one of $(5, 5)$. All of them have a stride of 1, and are followed by Relu Activation. All max-pooling layers have a window size of $(2, 2)$ and a stride of 2. After the classification of each frame in a batch of sequences is done and stacked, the tensor is sent to the LSTM cell. Then the output from the cell is flattened and converted to the classification result by a linear operation activated by softmax.

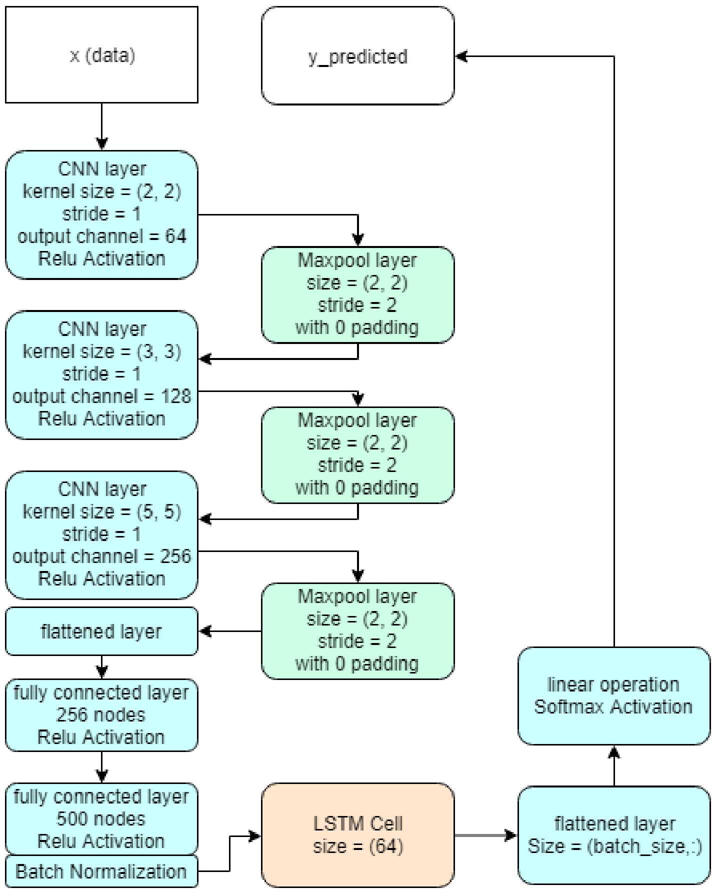

Figure 3: CNN + LSTM

3 Training

3.1 C3D

The most confusing result came from this model. Theoretically, this approach is simple, straight-forward and intuitive. However, no matter how I tune the parameters, including changing the learning rate from 0.1 to $1e-8$, the model always gave me a seemingly super-fast convergence with less than 5 epoches. The results given by the model were typically completely wrong for sure, while the loss jumped on the set $\{2.453, 1.453, 1.953\}$. I gave up trying on this model eventually.

3.2 VGG9

I did discuss with friends on this project but all of us ended up choosing different models. One of them suggested use VGG to classify each frame and try to combine the results. I tried to average the classification results in a sequence and it failed to converge. Then I tried the flattening-and-linear-algebra structure mentioned above and it converged really slowly. So I stopped the trials on this model.

3.3 LSTM

There are a number of similarities between PA4 and this project, including the sequences of images, the data format, etc. The most significant difference was that there should only be 1 classification result coming from each sequence other than for each frame. That requires more layers or linear algebras after the LSTM cell. This approach was actually the first method I thought of but was also the one that I worked on at last since I wanted to try new structures. Compared to the task we had for PA4, the model converges much slower on the UCF-11 data set. The learning rate was set to $1e-3$ with AdamOptimizer provided by TensorFlow. It did not help too much to make the learning rate higher or lower since the Adam has its own algorithms dealing with LR descending. Limited by the GPU memory, I set the batch size to 16 in order to squeeze up the computational power of it before the memory was squeezed up. The number of batches in an epoch was set to 256 and the number of epoches was set to 128. I also set an early stop when the accuracy on validation data set reached 75%. Validation data were composed of 160 sequences from the shuffled training data and were not used for training.

The initialization of the filters were generic: all elements in them were generated by following a normal distribution with a mean of 0 and the standard deviation of 0.1. The exceptions were the bias vectors, one of them participating in generating the inputs to the LSTM, the other converting the LSTM outputs into the classification result. They were initialized by assigning 0's to all their elements.

4 Results and Conclusion

4.1 Losses

The loss was defined as the cross-entropy softmax between the prediction vectors and the labels. The labels were converted to follow the one-hot encoding, so simply calling tensorflow function $tf.nn.softmax\_cross\_entropy\_with\_logits\_v2(logits = prediction, labels = y)$ did the job. The loss was calculated batch-wisely every batch but was shown every epoch. The losses calculated but were not shown were for training purposes and the ones calculated every epoch were for informing purposes. Typically, the mean value of the losses over one batch went from 2.4 to 0.5 as the validation accuracy went from 10% to the desired accuracy which was 75%.

4.2 Accuracy

The accuracy was calculated every epoch. At the end of each epoch, all 160 validation sequences were fed into the model batch by batch, and the overall accuracy was calculated as the average accuracy over all these batches. With the maximum number of epoches set to 128, the training would also stop once the accuracy reached 75%.

4.3 Confusion Matrix

The confusion matrix shows how well can the optimizer distinguish the action of a class from other ones. Each block (row, col) in this matrix shows the possibility that the model recognize action x as y.

4.4 Model

Due to the large size of the model, I decided to upload the .model files to Google Drive. The link below is to the folder where I saved the files. ECSE4850: Jiawei Xu Final Project Model Files

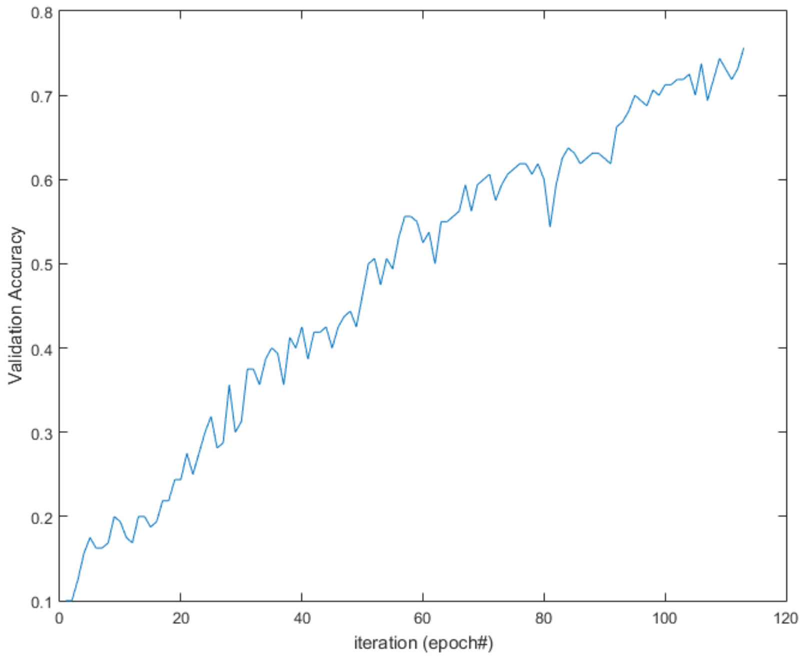

Figure 4: overall accuracies of the model

References

[1] Gül Varol, Ivan Laptev, and Cordelia Schmid. "Long-term temporal convolutions for action recognition". In: *IEEE transactions on pattern analysis and machine intelligence* 40.6 (2018), pp. 1510–1517.

[2] Chansung Park. *CIFAR-10 Image Classification in TensorFlow*. Apr. 2018. URL: https://towardsdatascience.com/cifar-10-image-classification-in-tensorflow-5b501f7dc77c.

[3] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).