Alexander Christner

May 4, 2019

chrisa3

4000

Introduction to Deep Learning Final Project Report

Model Architecture

The x input is a “batch_size” number of 30 image sequences, where each image has 64x64 pixels and 3 channels, resulting in an x input shape of [batch_size, 30, 64, 64, 3]. The y input is a vector of the x input’s corresponding classes between 0 and 10; a vector of shape [batch_size].

There are three sections of my architecture: a Convolutional Neural Network, a Recurrent Neural Network, and a Multilayer Perceptron. Each of the weights in these layers are initialized using Xavier initialization.

Each individual image in each sequence is passed through the CNN. To do this, I first reshape the x input from [batch_size, 30, 64, 64, 3] shape to a [batch_size*30, 64, 64, 3] shape; changing the first dimension from each batch to each image. The CNN takes an image of a sequence as an input and puts it through three sets consisting of a convolution layer, activation layer, and pooling layer. In each set, the convolution layer possesses 32 filters, each with a [5, 5] window size, a stride of 1, and no padding. Each activation layer uses the Relu activation function. Each pooling layer uses max pooling, has a window size of [2, 2], a stride of 2, and no padding. After each image is passed through the three sets, the result is a [batch_size*30, 4, 4, 32] tensor. I flatten this output to a [batch_size*30, 4*4*32] matrix in order to put it through a dense layer with 20 units, resulting in a [batch_size*30, 20] matrix; a vector of 20 weights for each image. Finally, I restore the original sequences by reshaping the output to a [batch_size, 30, 20] tensor; a vector of 20 weights for each image in each batch.

The output of the CNN is then put through an RNN. The RNN in my model is made of an LSTM cell with 30 units looped 30 times; one loop for each image in a sequence. This results in a tensor of shape [batch_size, 30, 30]; one vector of 30 weights for each of the 30 images in each batch.

The output of the RNN is put through a multilayer perceptron. First, I flatten the RNN output into a [batch_size, 30*30]; concatenating the 30 vectors in each sequence into one vector per sequence. My multiplayer perceptron consists of 2 dense layers. The first has 40 units, so it results in a [batch_size, 40] matrix. The second has 11 units, and takes the output if the first dense layer as the input, so it results in a [batch_size, 11] matrix; 11 class weights for each sequence.

Predictions are made by passing the output of the multilayer perceptron through the softmax function, then finding the argmax across the 11 resulting probabilities, creating in a [batch_size] vector of class predictions from 0 to 10.

The loss is found by passing the output of the multiplayer perceptron through the sparse_softmax_cross_entropy function, then adding L2 regularization with a constant of 0.0001.

Training is done by minimizing the loss using an Adam Optimizer (which handles the change in learning rate) with a starting learning rate of 0.001.

Model Justification

The number of variables in a sequence is extremely large; 30*64*64*3=368,640 numbers per sequence. Without reducing the number of variables early, later layers would each have to have an extremely large number of weights. The convolutional neural network at the beginning is designed to perform initial analysis on each individual image, while also reducing the number of weights that later layers have to use.

There are temporal dependencies within a sequence; that is to say, the meaning each image has is not merely dependent upon the image itself, but also upon the meaning of other images around it. The recurrent neural network with the LSTM cells is designed to model for these temporal dependencies between the images in a sequence when figuring out the meaning of each image.

The RNN results in a vector for each image in a sequence, but we need a way to map those vectors to a single vector of 11 probabilities. The multilayer perceptron is designed to map the output of the recurrent neural network to probabilities for each of the 11 classes. To do this, I flatten the 30 output vectors from the RNN into one vector, and the multilayer perceptron would learn how important each vector is to the class weights.

With those 11 probabilities, the predicted class is simply that with the highest of the probabilities. The probabilities are found using the softmax function on the 11 class weights found by the multilayer perceptron, and the predicted class is found with the argmax of those

Training Process

On each epoch of training, I use 100 random batches of 10 sequences from the training set to make prediction in order to estimate the model's accuracy (the training accuracy), and another 100 to estimate the model's loss (the training loss). I also use another 100 batches from the validation set to make predictions in order tot estimate the model's accuracy (the validation accuracy), and another 100 to estimate the model's loss (the validation loss).

Throughout training, I keep track of a counter, which starts at 0 in the beginning. On each epoch, the counter is incremented if the training loss increases, if the validation loss increases, if the total training accuracy decreases, and if the total validation accuracy increases (hence, it can increase by between 0 and 4 on each epoch).

If the counter reaches at least 25, if the max epoch (100) has been reached, or the validation accuracy has reached at least 90%, I consider the model to have converged, and stop training.

Otherwise, I use 400 random batches of 10 sequences to run the train step (described above) in order to further train the model. Then, I repeat another epoch.

Parameter Tuning

When deciding how many units each dense layer would have, how many each LSTM cell would have, and how many sequences a batch would contain, I started with larger numbers and decreased them until my computer could handle it in a reasonable amount of time.

When deciding the starting learning rate, the regularization constant, the weight initialization, convolution network properties, I just used what worked in previous projects and saw that it worked well here as well.

To decide the stopping condition, I let the training loop run for about a hundred iterations without stopping and watched how the accuracy and loss changed over time. I found it hard to get the model over 90% validation, but felt that was sufficient and left the minimum accuracy to stop there. When deciding how many times the accuracy or loss had to get worse for the model to stop, I looked for the point where the loss and accuracy consistently got worse and saw how many counts and how many epochs it took to get there, and I settled on 25 and 30 respectively.

Results

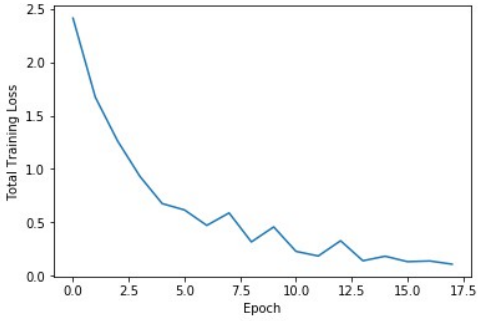
Final Training Loss: 0.108

Final Total Training Accuracy: 0.971

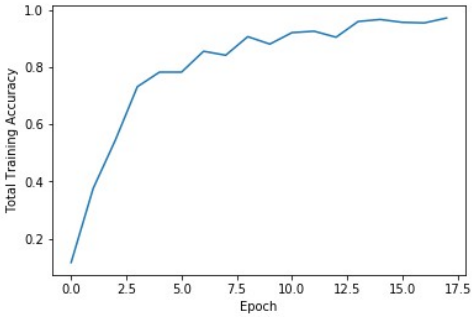
Final Training Confusion Matrix

Prediction Truth	Basketball shooting	Biking	Diving	Golf swinging	Horseback riding	Soccer juggling	Swinging	Tennis swinging	Trampoline jumping	Volleyball spiking	Dog walking
Basketball shooting	0.978	0.022	0	0	0	0	0	0	0	0	0
Biking	0	0.956	0	0	0	0.011	0.022	0.011	0	0	0
Diving	0	0	0.992	0	0	0	0	0	0	0	0.008
Golf swinging	0	0	0	1.0	0	0	0	0	0	0	0
Horseback riding	0	0	0	0.017	0.924	0	0.017	0	0	0.017	0.025
Soccer juggling	0	0	0	0	0	1.0	0	0	0	0	0
Swinging	0	0	0	0	0	0	0.985	0	0	0	0.015
Tennis swinging	0	0	0	0	0	0	0	1.0	0	0	0
Trampoline jumping	0	0	0	0	0	0.016	0	0	0.984	0	0
Volleyball spiking	0	0	0	0	0	0	0	0	0	1.0	0
Dog walking	0	0.011	0	0.011	0.011	0	0.066	0	0	0.022	0.879

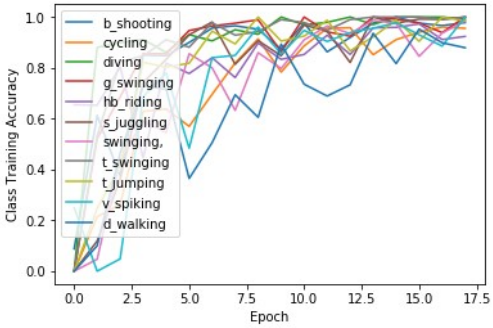
Training Loss



Total Training Accuracy



Class Training Accuracy



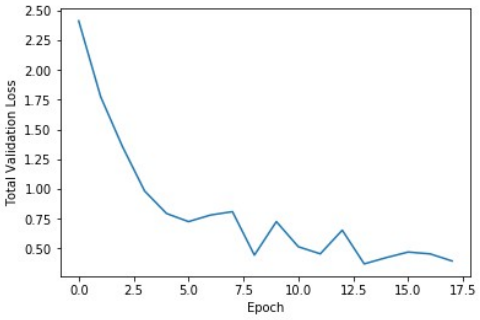
Final Validation Loss: 0.395

Final Validation Training Accuracy: 0.905

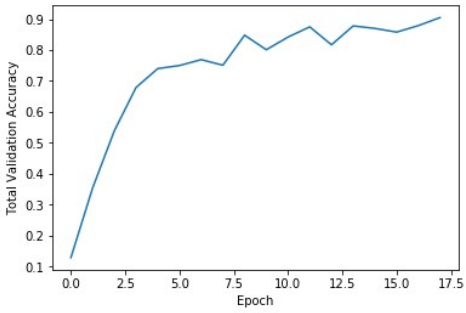
Final Validation Confusion Matrix

Prediction Truth	Basketball shooting	Biking	Diving	Golf swinging	Horseback riding	Soccer juggling	Swinging	Tennis swinging	Trampoline jumping	Volleyball spiking	Dog walking
Basketball shooting	0.955	0.03	0.15	0	0	0	0	0	0	0	0
Biking	0	0.889	0	0	0.011	0.022	0.067	0	0	0	0.011
Diving	0	0	0.96	0.01	0	0	0	0.01	0	0	0.02
Golf swinging	0	0	0	1.0	0	0	0	0	0	0	0
Horseback riding	0	0.017	0	0.026	0.793	0.017	0.026	0.026	0.017	0.052	0.026
Soccer juggling	0	0	0	0	0.009	0.963	0.028	0	0	0	0
Swinging	0	0.084	0.011	0.032	0	0.021	0.821	0	0	0.021	0.011
Tennis swinging	0	0	0	0	0	0	0	0.983	0	0	0.017
Trampoline jumping	0	0.045	0	0	0	0	0.06	0	0.896	0	0
Volleyball spiking	0	0.014	0	0.014	0	0	0.014	0	0	0.959	0
Dog walking	0	0.075	0	0.025	0	0.087	0.05	0	0	0.025	0.738

Validation Loss



Total Validation Accuracy



Class Validation Accuracy