**Convolutional Neural Network Structure**

We prepared a fully configurable Convolutional Neural Network function for the task of separating 32x32 RGB images into 10 classes (CIFAR-10 dataset). In this report, firstly we will describe the input and output relation of the prepared MATLAB function for generic CNN implementation. Secondly, the implementation details this function will be described with an emphasis on key points that we worked on to decrease the computational complexity.

1. **Generic CNN implementation**

The first step of our generic CNN development was to determine the types and scopes of the parameters. We decided that a CNN architecture which depends on only pooling layers (max or mean) would work for our project. Then we decided to prepare a generic CNN structure based on the following parameters that are grouped under the units of convolutional, fully connected and general properties of the CNN architecture.

1. Parameters related to the convolutional part of the CNN architecture:
   * Number of input channels
   * The size of each input channel (We assumed that the size of each channel is same.)
   * Number of convolutional layers and number of channels in each one of them.
   * Learning rate for each convolutional layer.
   * Zero padding size for each convolutional layer. (respective horizontal and vertical values are specified separately)
   * Kernel initialization information for each convolutional layer. (either one of normal distribution with mean and standard deviation or uniform distribution between two given values)
   * Pooling type for each convolutional layer (max or mean)
   * Pooling size for each convolutional layer (respective horizontal and vertical values are specified separately)
   * Convolutional bias type for each convolutional layer (tied or untied).
   * Activation function to be used in each convolutional layer (ReLU, tanh, logistic and softmax are available choices. ReLU is the most frequently used activation function.)
2. Parameters related to the fully connected part of the CNN architecture:
   * Number of output neurons
   * Number of hidden layers and number of neurons in each hidden layer
   * Learning rate for each fully connected layer
   * Weight initialization information for each fully connected layer. (either one of normal distribution with mean and standard deviation or uniform distribution between two given values)
   * Activation function to be used in each convolutional layer (ReLU, tanh, logistic and softmax are available choices. Softmax activation function is used in the output layer.)
   * Momentum coefficient for the whole fully connected network
   * Weight decay parameter for the whole fully connected network
3. Parameters related to general settings of the CNN
   * Number of epochs
   * Minibatch size
   * Train data
   * Train labels
   * Validation data
   * Validation labels
   * Test data
   * Test labels
   * Validation section (To indicated the currently processed fold in the k-fold cross validation process.)

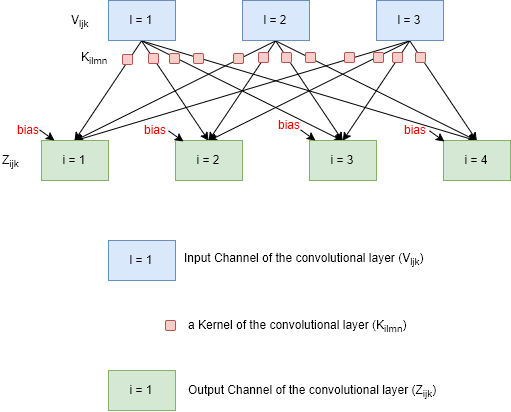
According to these parameters, we prepared our generic CNN architecture. A sample utilization of our generic CNN architecture is presented below.

|  |
| --- |
| % Convolutional Layers' Settings  input\_struct.conv\_input\_channel = 3;  input\_struct.conv\_input\_size = size(redChannel);  input\_struct.conv\_layer\_content = [8 10 10];  input\_struct.conv\_learning\_rate = [0.1 0.1 0.1];  input\_struct.conv\_zero\_padding\_size = [2 2 2];  input\_struct.conv\_kernel\_size = {[5,5] , ...  [5,5] , ...  [5,5]};  input\_struct.conv\_kernel\_init = {{'normal', 0 , 0.01} , ...  {'normal', 0 , 0.01} , ...  {'normal', 0 , 0.01}};  input\_struct.conv\_pooling\_type = {'max' , ...  'max' , ...  'max'};  input\_struct.conv\_pooling\_size = {[2,2] , ...  [2,2] , ...  [2,2]};  input\_struct.conv\_bias\_type = {'untied' , ...  'untied' , ...  'untied'};  input\_struct.conv\_activation = {'ReLU' , ...  'ReLU' , ...  'ReLU'};    % Fully Connected Layers' Settings  input\_struct.fulconn\_output\_neuron\_count = 10; % Fixed  input\_struct.fulconn\_hidden\_layer\_neuron\_content = [128];  input\_struct.fulconn\_learning\_rate = [0.1 0.1];  input\_struct.fulconn\_weight\_init = {{'uniform',-0.1 , 0.1},{'uniform',-0.1 , 0.1}};  input\_struct.fulconn\_activation = {'tanh','softmax'};  input\_struct.fulconn\_momentum\_coef = 0.9;  input\_struct.fulconn\_weight\_decay = 0.0001;    % Network's General Settings  input\_struct.epoch\_num = 20;  input\_struct.minibatch\_size = 20;  input\_struct.train\_data = train\_data;  input\_struct.train\_label = train\_label;  input\_struct.validation\_data = validation\_data;  input\_struct.validation\_label = validation\_label;  input\_struct.test\_data = test\_data;  input\_struct.test\_label = test\_label;  input\_struct.validation\_section = validation\_section;      % Run the algorithm  [output\_struct] = implement\_cnn\_algorithm(input\_struct); |

1. **Implementation Details and Computational Complexity Reduction.**

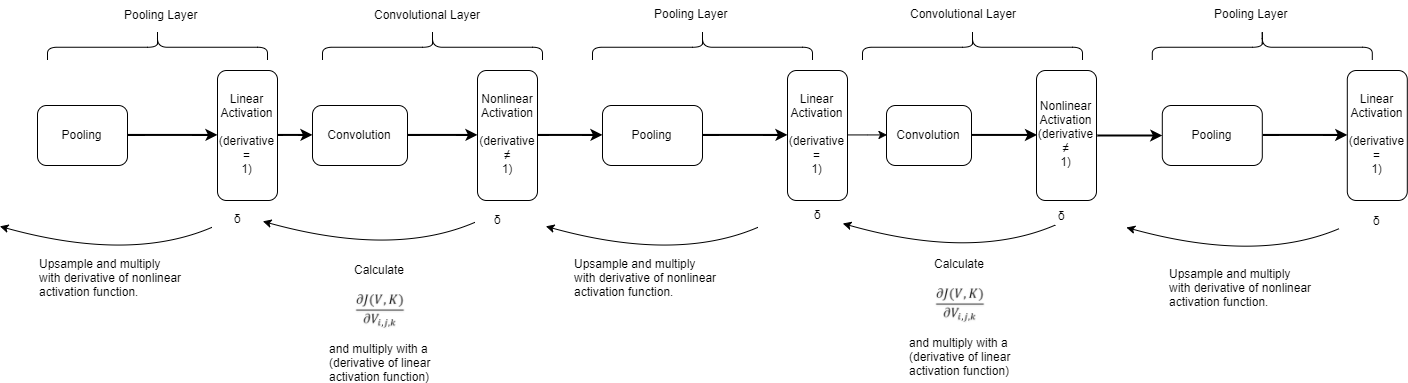
We already know how to implement a fully connected layer. When a fully connected network is connected to a convolution one, the functionality of the fully connected one is not changed. Its input is supplied by the convolutional part and in the backpropagation operation, the error gradient obtained in the input of the fully connected layer is transferred to the convolutional part.

In the implementation of the convolutional part, we have input and output channels and kernels that connect them. The visualization of convolutional network units is presented on the figure below. Based on this visualization, forward and backward propagation are formulated as follows [1].



**Forward Propagation of Input Data (Summation of Convolutions without Flipping the kernel)**

**Backward Propagation of Gradients**



**Weight updates**

(for untied bias)

(for tied bias)

Once we learned the formulation of CNN basics, we started to implement them on them Matlab code. As the code gets bigger, the total processing time gets longer. When we analyzed the parts of the code that consumed most of the time, we observed that pooling and backpropagation of error gradient through a convolution layer takes most of the time. It was due to the fact that we implemented these operations just by using for loops. Thus, we thought that optimizing these operations will be an important development, especially for long computations. Firstly, in order to avoid utilization of for loop in looping operation, we found a solution based on “accumarray” command of Matlab. Briefly, for max pooling, we specified pool regions with subarrays of different numbers and accumarray command found us the max value in that region and returned the result to us. Secondly, to reduce the time required by the backpropagation, we tried to find a way to express this big summation as an optimized operation. We found that it can be expressed as a convolution operation. These two refinements increased our processing speed considerably.

References

[1] “Deep Learning,” *Deep Learning*. [Online]. Available: http://www.deeplearningbook.org/. [Accessed: 08-Jan-2018].