ECE 464 / 564 Project : Fall 2022:
Deep Neural Network

Change Log

- 1/8/22 original spec

Reference:

- Sze et.al., "Efficient Processing of Deep Neural Networks", Morgan & Claypool https://catalog.lib.ncsu.edu/catalog/NCSU5013375

This project is to be conducted individually. You can collaborate on the paper version of the design, including discussion of ideas, design approach, etc. However, you are forbidden to share code or to reuse code of others. We will be running code comparison tools on your submitted code.

EOL students are expected to do the ECE 464 project.

ECE 564: Your task is to implement a multi-stage neural network, including a convolutional layer, a fully connected layer and a max pooling layer. On campus ECE 564 students can limit themselves to the ECE 464 project if they choose but at the cost of a 25% penalty.

ECE 464 and EOL (students in section 601) students: Your task is to implement a fixed size single stage of a convolutional neural network.

Artificial Neural Network (ANN)

An ANN is a multi-level network that can perform inference. It's composed, in part of a set of neurons per the figure below [Sze].

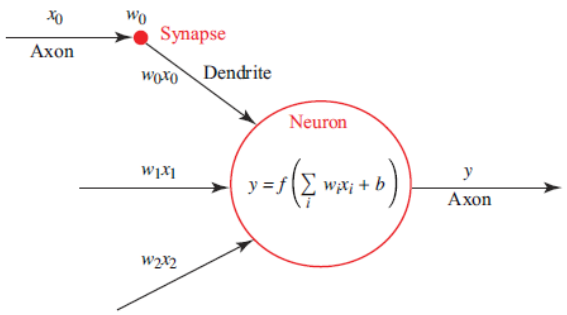
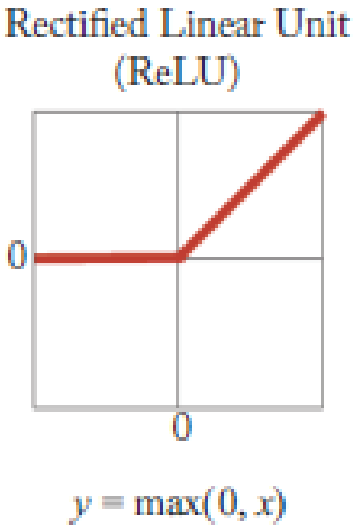


Figure 1.2: Connections to a neuron in the brain. x_i, w_i, $f(\cdot)$, and b are the activations, weights, nonlinear function, and bias, respectively. (Figure adapted from [4].)

The input to the neuron is the weighted sum of inputs Σ w_i*x_i, where x_i are the inputs and w_i are the weights. In this project, all variables are 8-bit signed integers. Note, x_i can only take on positive values. Computations are to saturate, not overflow. With 8-bits you can represent integers from -128 to +127 digital. +127 is represented as 0111_1111 or 7F.

The output of the sum of weights is then passed to an activation function f. This is non-linear function and represents an important reason why ANNs are so effective. We will use the ReLu activation function, illustrated below [Sze]. In this project there is no offset b.



Deep Neural Network

A deep neural network consists of multiple stages, each containing many neurons. An example is given below. In part it consists of convolutional stages, pooling layers and fully connected layers. [Sze]

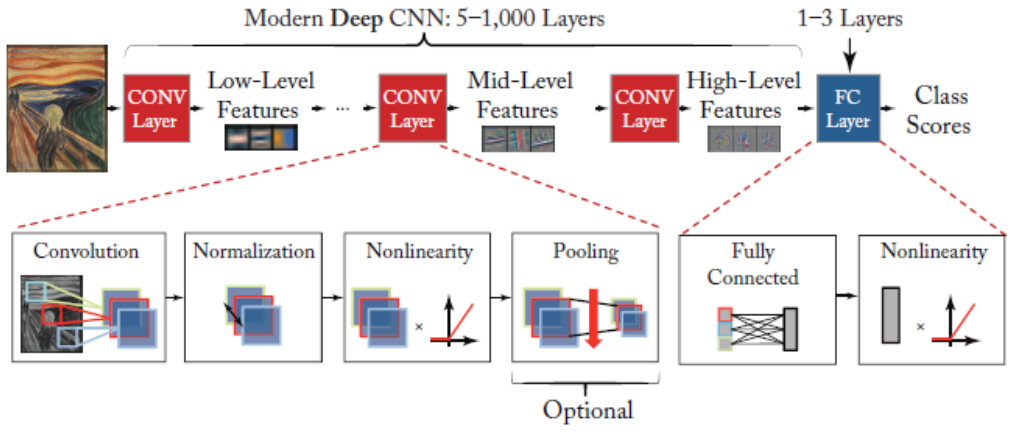


Figure 2.7: Convolutional Neural Networks.

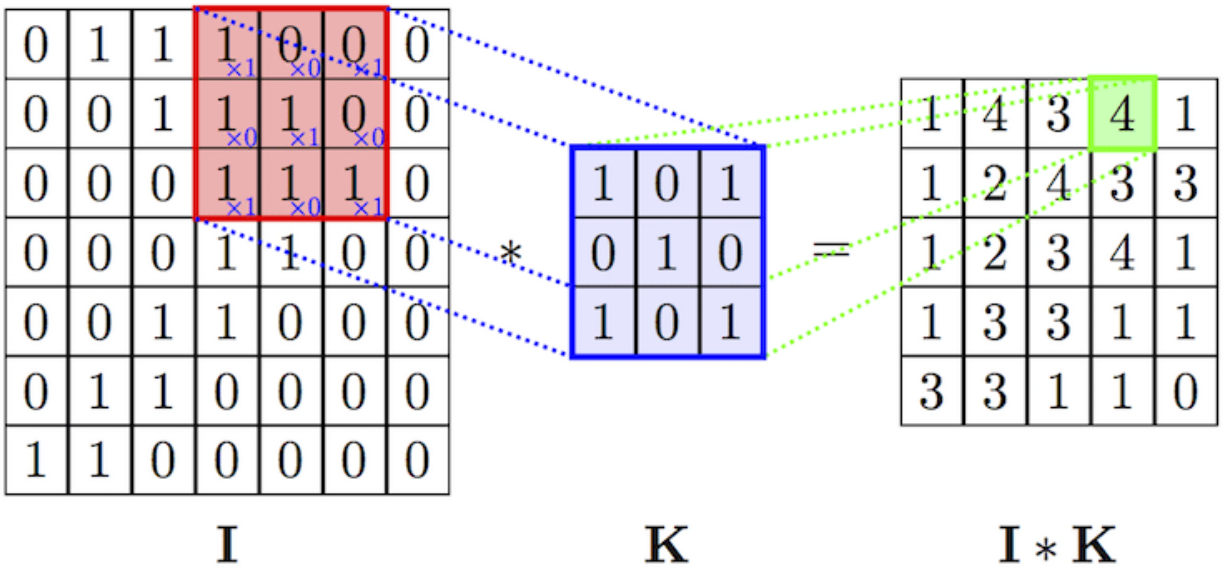
Convolutional Layer

In a convolutional neural network layer, the weight matrix, called a kernel, is smaller than the input matrix, and an output matrix is produced with a size smaller than the input.

In a convolutional stage the kernel is multiplied by a subrange of the input to produce the feature map. For example, using row major ordering, element [0,1] of the feature map is produced by the following computation

F[0,1] = f(i[0,1]*k[0,0]+ i[0,2]*k[0,1] + i[0,3]*k[0,2] + .. i[2,3]*k[2,2])

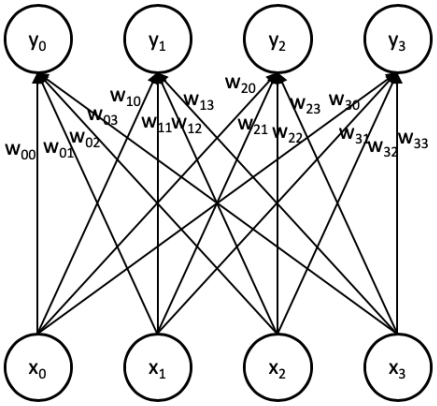
An example is illustrated below. (Taken from https://www.researchgate.net/publication/324165524_Detection_and_Tracking_of_Pallets_using_a_Laser_Rangefinder_and_Machine_Learning_Techniques/figures?lo=1) i.e. the kernel strides across and down the input matrix doing a complete calculation for each position.



Full Connected Layers

In fully connected layers, the outputs y are a sum of w_i times x_i across all the inputs. The output of each neuron will be the dot products of the input vector and weight vector which is then passed through an activation function as shown in the following equation, where $\emptyset(v_i)$ is the activation function.

$$y_k = \emptyset(w_0x_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n)$$



$$\begin{bmatrix} w_{00} & w_{01} & w_{02} & w_{03} \\ w_{10} & w_{11} & w_{12} & w_{13} \\ w_{20} & w_{21} & w_{22} & w_{23} \\ w_{30} & w_{31} & w_{32} & w_{33} \end{bmatrix} \mathbf{x} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$\begin{bmatrix} \emptyset(v_0) \\ \emptyset(v_1) \\ \emptyset(v_2) \\ \emptyset(v_3) \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

As we start to build up a layer of fully connected layer of the neural network, we can create a collection including all weight vectors and arrange (append and transpose) them into a matrix. With this weight matrix form, we can use matrix multiplication to write the computation as the evaluation above.

Pooling Layers

Pooling reduces the matrix size by taking the max, min or average of each region of the matrix. For example, max pooling each 2x2 region of a 6x6 array produces a 3x3 array. Max and average pooling are illustrated below [Sze].

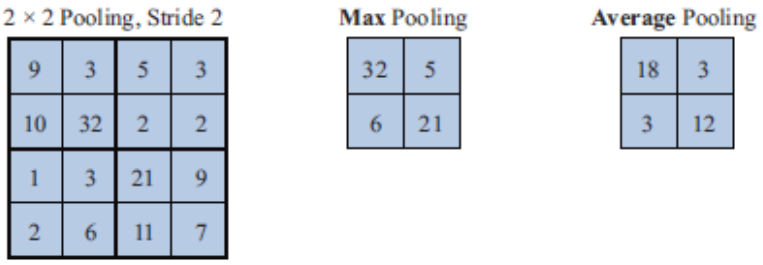


Figure 2.5: Various forms of pooling.

Project Specifications – ECE 464

- Inputs are a 16x16 8-bit per input array and is stored in the input SRAM. The inputs are signed but will only take on positive values.
- The kernel matrix is 3x3 8-bit array and is stored in the weight SRAM. The weights are signed and can be positive or negative.
- The output will be a 14x14 array and your design will load it into the output SRAM. The outputs will be 8-bits wide but will only take on positive values.

You will implement a single stage of a convolutional neural network with a ReLu activation function. You will need a multiplier and accumulator and to implement the ReLu function. The multiplier will have two 8-bit inputs and a 16-bit output. The accumulator will be 20 bits wide. This will ensure that there is no overflow. The result of the ReLu function will be saturated to 8 bits, i.e. Any value greater than 127 will be 127.

The IO are as follows:

- Interfaces to three SRAMs as described above. Behavior is described below.
- clk; Clock from the test fixture. Make sure this corresponds to the name of the clock in the synthesis script.
- reset_b; Reset from the test fixture. This will go active low at the start of the simulation run.
- dut_run; This will be set high by the test fixture when it is ready for a new run to start.
- dut_busy; This is an output of your design and will be high when the hardware is busy. It is to be set low by the reset. It will go high one cycle after dut_run goes high and stay high until one cycle after the last output is sent to the output memory.

In the 464 project, only one sample is run at a time. When dut_run goes high, the needed operations are performed for the one example stored in the SRAMs and the outputs written to the results SRAM.

We will give you four sets of files containing sample inputs and expected outputs. Two will be given with the project spec. Note these have to be stored in the subdirectories input_0 and input_1 respectively. Two more will be given just before the project demo. These are held back so as to prevent you from hard wiring your code for the provided inputs.

The organization of the SRAMs will be so as to contain packed values (two per 16-bit word) in row-major order.

The weight SRAM will be organized as follows:

Addr	Contents
0000	w00 w01 // i.e. w00 is in the high order bits
0001	w02 w10
0002	w11 w12
0003	w20 w21
0004	w22 00

The input SRAM will be organized as follows:

Addr	Contents
0000	x00 x01
…	

0007 x0,14 x0,15
0008 x10 x11
…
003F x15,14 x15,15

The output SRAM will be organized as follows:

Addr Contents
0000 o00 o01
0000 o02 o03
…
0031 o13,12 o13,13

ECE 564 Project

- Inputs are a NxN 8-bit per entry input array and is stored in the input SRAM. The inputs are signed but will only take on positive values. N will be provided in the memory for each problem. N will be a power of 2 and no larger than 64.
- The kernel matrix is 3x3 8-bit kernel array and is stored in the weight SRAM. The weights are signed and can be positive or negative.
- An ((N-2)/2)*((N-2)/2) weight matrix for the fully connected layer. This will be stored in the weight SRAM after the kernel.
- The output will be sized as ((N-2)/2)*((N-2)/2) array and your design will load it into the output SRAM. The outputs will be 8-bits wide but will only take on positive values.

You will implement 3 stages, a single stage of a convolution with a ReLu function, a single stage of max pooling, and a single stage of a fully connected layer with a ReLu function.

You will implement a single stage of a convolutional neural network with a ReLu activation function. You will need a multiplier and accumulator and to implement the CNN function. The multiplier will have two 8-bit inputs and a 16-bit output. The accumulator will be 20 bits wide. This will ensure that there is no overflow. The result of the ReLu function will be saturated to 8 bits, i.e. Any value greater than 127 will be 127.

The resulting matrix from this stage will have size (N-2)*(N-2) and have 8-bit entries.

You will implement a MaxPooling function based on the max in each 2*2 subarray. The input to the MaxPooling layer will be the output of the convolutional layer. No ReLu is needed. The output from this stage will be an ((N-2)/2)*((N-2)/2) array.

This will feed a last fully connected layer. The output from this stage will be an ((N-2)/2)*((N-2)/2) array which will be written to the output SRAM. The multiplier will have two 8-bit inputs and a 16-bit output. The accumulator will be 20 bits wide. This (together with control over the

weights in training) will ensure that there is no overflow. The result of the ReLu function will be saturated to 8 bits, i.e. Any value greater than 127 will be 127.

The IO are as follows:

- Interfaces to three SRAMs as described above. Behavior is described below.
- Interface to a fourth SRAM. You can use this how you please.
- clk; Clock from the test fixture. Make sure this corresponds to the name of the clock in the synthesis script.
- reset_b; Reset from the test fixture. This will go active low at the start of the simulation run.
- dut_run; This will be set high by the test fixture when it is ready for a new run to start.
- dut_busy; This is an output of your design and will be high when the hardware is busy. It is to be set low by the reset. It will go high one cycle after dut_run goes high and stay high until one cycle after the last output is sent to the output memory.

In the 564 project, the weights are fixed for each run. However, multiple input sets will be provided. You are to keep running the inputs until the size entry in the memory takes on the value FFFF. The first entry will be a valid array size, not FFFF. When dut_run goes high, the needed operations are performed for the one or more examples stored in the SRAMs and the outputs written to the results SRAM.

We will give you four sets of sample inputs and expected outputs. Two will be given with the project spec. Note these have to be stored in the subdirectories input_0 and input_1 respectively. Two more will be given just before the project demo. These are held back so as to prevent you from hard wiring your code for the provided inputs.

The organization of the SRAMs will be so as to contain packed values (two per 16-bit word) in row-major order.

The weight SRAM will be organized as follows. The convolutional kernel is stored first.

Addr	Contents
0000	k00 k01 // i.e. k00 is in the high order bits
0001	k02 k10
0002	k11 k12
0003	k20 k21
0004	k22 00
0005	w00 w01 // weights for fully connected matrix
…	
?	last pair of weights – depends on N

The input SRAM will be organized as follows. An example is given where the first input matrix is 16x16.

3x3
7x7
15x15

Addr Contents
0000 00 N1 // note 0s stored first
0001 x00 x01
…
0008 x0,14 x0,15
0009 x10 x11
…
0040 x15,14 x15,15
0041 00 N2 // if this is FF FF stop processing
0042 x00 x01

The output SRAM will be organized in a way where zeros are appended at the end of every set of output corresponding to each set of N *N inputs.

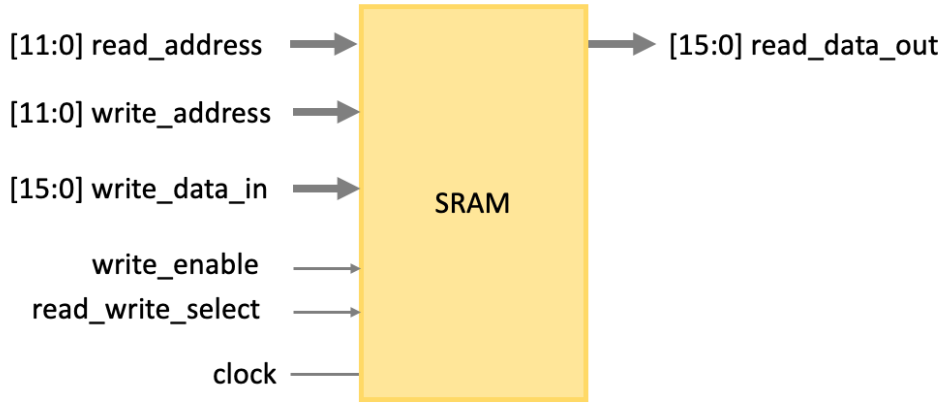
Memory structure

For the ECE 564 project, there are four SRAMs provided, one for weights, one for input data, one for output results, and one for a scratch pad. Each memory is 16 bits wide, and is 16-bit word addressable. The weight matrices won't change during a run. Multiple inputs will be presented though the input SRAM.

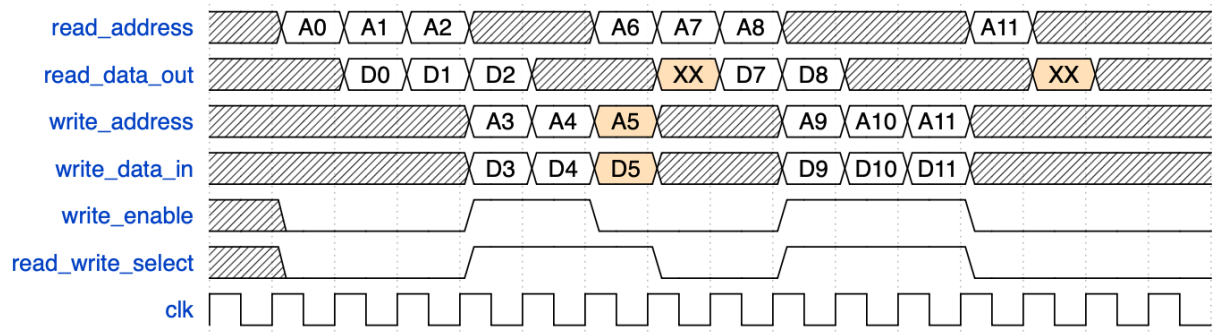
Hierarchy

I suggest that you organize your design as one module. Make sure to modify modname in the synthesis script as needed. If you have a hierarchy, specify the top module as modname.

SRAM interface



The SRAM is word addressable and has a one cycle delay between address and data. When writing to the SRAM, you would have to set the “write_enable” to high. The SRAM will write the data in the next cycle.



As shown in the example above, since “write_enable” is set to low when A5 and D5 is on the write bus, D5 will not be written to the SRAM. Also, because “read_write_select” is set to high, the read request for A6 will not be valid.
Note that the SRAM cannot handle consecutive read after write (RAW) to the same address (shown as A11 and D11 in the timing diagram). You would have to either manage the timing of your access, or write the data forwarding mechanism yourself. As long as the read and write address are different, the request can be pipelined.