EE113D - Digital Signal Processing Design

MiniProject: Convolutional Neural Network for MINST

Table of Contents

Objective

The objective of this mini project was to implement and train a Convolutional Neural Network in Python on Google Colab using the TensorFlow library, then port it into CubeIDE to be run on the Nucleo H7 board. Training models on the cloud saves a significant amount of time due to the limited computational resources on embedded systems. Since the H7 board has only a small amount of computational resources such as RAM and memory, this involved reducing the number of parameters in the Convolutional Neural Network provided in the base code so that the model was small enough to be run without issues on the H7 board while maintaining good accuracy. The MNIST dataset containing 28x28 pixel images of digits from 0-9 was used for the model to classify.

Description of CNN Model

Before training the neural network, the biases were initialized to 0 and the weights were deterministically randomized via a seeded random number generator. During training, the weights and biases are updated through back propagation in response to the error calculated after each attempted input classification. After training was completed the weights and biases were downloaded onto a USB drive so that the USB drive could be connected to the H7 board for the H7 to read the weights and biases to be used on the model on the H7. It was possible to use the pre-trained weights and biases from the Ipynb by keeping the dimensions of the model on the H7 the same as the model in the Ipynb.

The number of parameters in the default model was 61,514. In our slimmed down model, the number of parameters was 2,158. The accuracy of the default model was 98.41% while the accuracy of the slimmed down model was 97.25%. This is a significant reduction in size while still maintaining minimal loss in accuracy. Our final CNN model was composed of 2 convolution layers, 2 max pooling layers, and a dense layer.

Our convolutional layers implement a 3x3 kernel with a 1-pixel stride that detects features within each image via cross-correlation. We used a 1-pixel-wide periphery as padding to ensure that the dimensions of the output of the 2 convolutional layers remained the same dimensions as the input. The convolutional layer uses a non-linear ReLu activation function, which is a piecewise function that converts any negative neuron output value to 0, and keeps any positive value to send as input to the subsequent maxpool layer.

The output of the maxpool layer is generated by taking the maximum value of each 2x2 section of the input and putting this value in a smaller corresponding output matrix. The output matrix is ¼ the size of the input matrix since both the height of the input and width of the input is reduced by ½ during the maxpool operation. The maxpool layer reduces the number of parameters that subsequent layers need to operate on.

The dense layer is a fully connected layer that performs a forward array multiplication on the input layer to generate a 1x10 vector. Each value of the 1x10 vector is exponentiated, accumulated, and divided by the accumulated sum (softmax) and corresponds to each possible classification of the input, which in this case is the digits 0-9, and contains the corresponding score of each classification based on the features detected in the previous layers. We then loop over the scores to find the maximum and print out the corresponding class with its score.

Ipynb CNN Code

Default Model
    accuracy : 0.9811
    parameters : 61,514 {float}

1st attempt model

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1), padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(10, activation='softmax'))

params: 25258
acc: 0.9841
```

2nd attempt model

```
model = models.Sequential()
model.add(layers.Conv2D(16, (3, 3), activation='relu', input_shape=(28, 28, 1), padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(16, (3, 3), activation='relu', padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(10, activation='softmax'))

params: 10330
acc: 0.9790
```

---
final model

```
model = models.Sequential()
model.add(layers.Conv2D(4, (3, 3), activation='relu', input_shape=(28, 28, 1), padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(4, (3, 3), activation='relu', padding='same'))
```

```
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(10, activation='softmax'))
```

params: 2158
acc: 0.9725

H7 CNN Code

```
float * dense_layer(float * weights, float * biases, float * input, int
input_size){
	//get dimensions of input
	int N = input_size; //should be 784 wide array
	int out_size = 10;
	//weights are sized: 784 x 10
	//simply do a forward array multiplication (1x784) * (784*10) = (1*10)

	float * result;
	result = (float *)malloc(out_size*sizeof(float));
	int i, j;
	float sum = 0.0;
	//matrix multiplication with softmax built in
	for(i=0; i < out_size; i++){
		result[i] = biases[i];
		for(j=0; j < N; j++){
			result[i] += input[j] * weights[out_size*j + i];
		}
		result[i] = exp(result[i]);
		sum += result[i];
	}
	//final part of softmax
	for(i=0; i < out_size; i++){
		result[i] = result[i] / sum;
	}
	return(result);
}

float * conv_layer(float*weights, float*biases, float* input, int channels,
int input_shape, int filters){
	//input has shape H, W, C (height, width, channels)
	//weights have shape HH, WW, C, F (filter height, filter width,
channels, filter number)
	//biases have shape F
	//stride of 1
	//pad of 1

	int C = channels;
	int H = input_shape;
	int W = input_shape;

	int F = filters; //number of filters
	int HH = 3; //kernel height
	int WW = 3; //kernel width
```

```
	//padding input
	float* padded_input;
	padded_input = (float*) malloc(sizeof(float)*(H+2) * (W+2)*C); //make
sure to delete this after function finishes calculating
	int i, j, k;
	for(i = 0; i < H+2; i++){
		for(j=0; j<W+2; j++){
			for(k=0; k<C; k++){
				if(j!=0 && j!=H+1 && i!=0 && i!=W+1){
					padded_input[(H+2)*C*i + C*j + k] =
input[C*H*(i-1) + C*(j-1) + k];
				}
				else {
					padded_input[(H+2)*C*i + C*j + k] = 0.0;
				}
			}
		}
	}

	//dimension of result: H, W, F (since we are same padding)
	//each filter gets a bias added; not each channel (same bias for each
channel)
	//addressing 1d weights by filter number and channel number:
	//weights4d[i][j][chan][filt] == weights1d[i*n_chan*n_filt*width +
j*n_chan*n_filt + n_filt*chan + filt]

	float*output;
	output = (float*) malloc(sizeof(float)*F*H*W);

	//convolution with built in relu
	int fil, chan, x, y;
	for(x=0; x<H; x++){
		for(y=0; y<W; y++){
			for(fil = 0; fil<F; fil++){
				output[W*F*x + F*y + fil] = biases[fil];
				for(chan=0; chan< C; chan++){
					for(i=0; i <HH; i++){
						for(j=0; j <WW; j++){
							output[W*F*x + F*y + fil] +=
weights[F*C*WW*i + F*C*j + F*chan +fil] * padded_input[(W+2)*C*(i+x) +
C*(j+y) + chan];
						}
					}
				}
				if(output[W*F*x + F*y + fil] < 0.0){
					output[W*F*x + F*y + fil] = 0.0;
				}
			}
		}
	}
	free(padded_input);
	return(output);
}

float * maxpool_layer(float* input, int channels, int input_shape){
	//assuming 2x2 window
	//output shape should be (H/2) * (W/2) * chan
```

```
      int H = input_shape;
      int W = input_shape;
      int HH =(int) H/2;
      int WW =(int) W/2;
      int C = channels;

      float * output;
      output = (float*) malloc(sizeof(float)*(HH)*(WW)*C);

      int x, y, chan, i, j;
      for(x=0; x<HH; x++){
            for(y=0; y<WW; y++){
                  for(chan=0; chan< C; chan++){
                        output[WW*C*x + C*y + chan] = 0.0;
                        for(i=0; i <2; i++){
                              for(j=0; j <2; j++){
                                    if(output[WW*C*x + C*y + chan] <
input[W*C*(2*x+i) + C*(2*y+j) + chan]){
                                          output[WW*C*x + C*y + chan] =
input[W*C*(2*x+i) + C*(2*y+j) + chan];
                                    }
                              }
                        }
                  }
            }
      }
      return(output);
}

//code in main
//weights w1, w2, w3, b1, b2, b3 are read in using read_txt function
l1out = conv_layer(w1, b1, input_image, 1, 28, 4);
l2out = maxpool_layer(l1out, 4, 28);
l3out = conv_layer(w2,b2, l2out, 4, 14, 4);
l4out = maxpool_layer(l3out, 4, 14);
l5out = dense_layer(w3, b3, l4out, 196);

int max =0;
int max_index = 0;
for(int i =0; i<10; i++){
      printf("Probability of class %d: %f \n",i, l5out[i]);
      if(l5out[i] > max){
            max = l5out[i];
            max_index = i;
      }
}

printf("Predicted class is %d with probability %f\n", max_index, max);
```

Results

Ground Truth	Prediction	
	numX.bmp	numX_t.bmp
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	3	5
6	8	5
7	7	2
8	3	8
9	7	8
# Correct	13	
# Wrong	7	
Accuracy	65%	

Sample output window (input image is 1)

```
Port 0 ×
The input image is shown:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 207 207 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 16 239 255 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 207 239 48 255 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 191 239 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 239 207 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 255 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 16 207 255 255 255 255 255 255 255 255 255 255 255 255 191 0 0 0 0 0 0
 0 0 0 0 0 0 0 175 48 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 Probability of class 0: 0.000000
Probability of class 1: 0.881651
Probability of class 2: 0.000082
Probability of class 3: 0.058887
Probability of class 4: 0.000000
Probability of class 5: 0.003745
Probability of class 6: 0.000001
Probability of class 7: 0.000000
Probability of class 8: 0.055633
Probability of class 9: 0.000001
Predicted class is 1 with probability 0.881651
```

Discussion

This project reveals that MNIST is a relatively easy task for a convolutional neural network. A relatively large NN model is able to maintain accuracy as scale increases, without overfitting. This explains why the provided model was able to be significantly reduced in size while maintaining accuracy. The test accuracy of the model on the H7 is only 65%, which can be explained by the smaller testing set (only 20 testing images). If we want a testing accuracy similar to the one achieved in the Python notebook, then we would need to either increase the size of the test set on the H7 or use a model with more weights (or both). In implementing the convolutional forward pass, the biggest problem involved indexing the input and weights. Since the input image and weights are a flat 1D array, then we needed to figure out a formula for indexing the arrays as if they were 4D so that a 2D convolution could be performed. While there was the option of reshaping the arrays into a proper 4D array, that method involves dynamically allocating (and subsequently deallocating) a quadruple pointer which is much more complicated and less efficient compared to figuring out the 1D-to-4D indexing formula. Afterwards, implementing the calculations for each layer was much simpler and only involved figuring out how to do dot products in C (which can be done with multiple for loops). In comparison to Python, implementing the CNN was much more complicated but also helped in understanding the underlying mathematical calculations done by each forward pass, which further improved our understanding of how CNNs worked.