KNN-based digit recognition

1. HLS C-sim/Synthesis/Cosim (Screenshot + brief intro) (1%)

KNN(K-Nerarest Neighbors)又稱為 K-近鄰演算法，其原理為找到距離最近的 K 個鄰居，進行投票，決定類別。

Step1 計算距離：計算每個鄰居與自己的距離

Step2 進行投票：最接近的 K 個鄰居，分別屬於哪些類別

Step3 決定類別：選擇最多人有的類別，來決定本身的類別

Ex: K 的選擇(最好選擇奇數，以免造成同票的情況)

◎圖 1 K=3 星星會找到最近的三個點，而向上水滴佔多數，因此被歸類為向上水滴
K=5 星星會找到最近的五個點，而向下水滴佔多數，因此被歸類為向下水滴

◎圖 2 K=4 星星會找到最近的四個點，此時發現票數一樣，因此最好不要使用偶數的 K 值。
解決方法採用加權投票法，即越靠近本身的鄰居會擁有較高權重，因此被歸類為向上水滴

Reference: https://www.youtube.com/watch?v=RWvM-9V3UzY&ab_channel=PyInvest

```
14 bit4 digitrec( digit input )
15 {
16   #include "training_data.h"
17
18   // This array stores K minimum distances per training set
19   bit6 knn_set[10][K_CONST];
20
21   // Initialize the knn set
22 DIGITREC_INIT_LOOP_OUTER:
23   for ( int i = 0; i < 10; ++i )
24 DIGITREC_INIT_LOOP_INNER:
25     for ( int k = 0; k < K_CONST; ++k )
26       // Note that the max distance is 49
27       knn_set[i][k] = DIGIT_SIZE+1;
28                                                    測試每個鄰居
29   for ( int i = 0; i < TRAINING_SIZE; ++i ) {
30 DIGITREC_PROC_LOOP_INNER:
31     for ( int j = 0; j < 10; j++ ) {
32
33       // Read a new instance from the training set
34       digit training_instance = training_data[j * TRAINING_SIZE + i];
35       // Update the KNN set
36       update_knn( input, training_instance, knn_set[j] );
37     }
38   }
```

Step1 計算距離

```
58 void update_knn( digit test_inst, digit train_inst, bit6 min_distances[K_CONST] )
59 {
60   // compute the distance between test_inst and train_inst
61   // if the distance is smaller than some elements in min_distances[] then
62   // update the array
63   bit6 dist = 0;
64   digit difference = test_inst ^ train_inst;
65
66   // compute the distance
67 UPDATE_DIFF_LOOP:
68   for ( int i = 0; i < DIGIT_SIZE; i++ ) {
69     if ( difference & 0x1 )
70       dist++;
71     difference = difference >> 1;
72   }                                                 計算距離
73
74   // update the min_distances array
75 UPDATE_DIST_LOOP:
76   for ( int i = 0; i < K_CONST; i++ )
77     if ( min_distances[i] > dist ) {
78       min_distances[i] = dist;
79       break;                                        依照 K 值選擇最近的 K 個鄰居
80     }
81 }
```

```
96 bit4 knn_vote( bit6 knn_set[10][K_CONST] )
97 {
98   bit6 cur_dist[K_CONST];
99   bit4 cur_digit[K_CONST];
100
101  bit4 vote[10];
102  bit4 max_voted_digit = -1, max_vote = 0;
103
104 VOTE_INIT_DIST_LOOP:
105   for ( int i = 0; i < K_CONST; i++ ) {
106     cur_dist[i] = DIGIT_SIZE+1;
107     cur_digit[i] = -1;
108   }
109
110 VOTE_INIT_VOTE_LOOP:
111   for ( int i = 0; i < 10; i++ )
112     vote[i] = 0;                                   進行投票並且選擇最高票作為類別
113
114 VOTE_MIN_DIST_DIGIT_LOOP:
115   for ( int d = 0; d < 10; d++ )
116 VOTE_MIN_DIST_CONST_LOOP:
117     for ( int k = 0; k < K_CONST; k++ ) {
118       // compare knn_set[d][k] with current results
119 VOTE_MIN_DIST_CUR_CONST_LOOP:
120       for ( int cur_k = 0; cur_k < K_CONST; cur_k++ )
121         if ( cur_dist[cur_k] > knn_set[d][k] ) {
122           cur_dist[cur_k] = knn_set[d][k];
123           cur_digit[cur_k] = d;
124           break;
125         }
126     }
127
128 VOTE_CALC_VOTE_LOOP:
129   for ( int i = 0; i < K_CONST; i++ )
130     vote[cur_digit[i]]++;
131
132 VOTE_FIND_MAX_VOTE_LOOP:
133   for ( int i = 0; i < 10; i++ )
134     if ( vote[i] > max_vote ) {
135       max_vote = vote[i];
136       max_voted_digit = i;
137     }
138
139   return max_voted_digit;
140 }
```

Step2 進行投票
Step3 決定類別

◎圖 3 Baseline Code(digitrec.cpp)

The Screenshot is for the Baseline Code:

```
Overall Error Rate = 6.11%
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

◎圖 4 C-sim

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.379 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
993837	1101897	9.938 ms	11.019 ms	993837	1101897	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	315	-
FIFO	-	-	-	-	-
Instance	0	-	109	479	0
Memory	96	-	12	3	0
Multiplexer	-	-	-	188	-
Register	-	-	175	-	-
Total	96	0	296	985	0
Available	280	220	106400	53200	0
Utilization (%)	34	0	~0	1	0

◎圖 5 C-Synthesis

Cosimulation Report for 'digitrec'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	1083029	1083202	1083343	1083030	1083203	1083344

Export the report(.html) using the Export Wizard

◎圖 6 Cosim

2. Improvement – throughput, area (1%)

測試不同的 K 來觀察對 Error Rate 的影響:

K	Error Rate(%)
2	10.6%
3	6.11%
4	6.11%
5	6.67%
6	5.56%
7	7.22%
8	7.22%

結論: K 值是一個重要的參數，盡量避免使用偶數的 K 值，且 K 值也不能設定太小(noise)或太大(non-relation)，根據經驗法則 K 值低於樣本數的平方根。依照不選偶數的原則下，我們選擇 K=3 進行以下優化。

Solution1:Array Partition

從測試每個鄰居並選擇前三近的這段 Code 中，可以看出我們必須在 update_knn 時，會用到 knn_set 這個 Matrix，但是因為一個 Matrix 一次只能取一個資料，所以我們可以利用 Array Partition 的方式，以一次取多個資料。

```
14 bit4 digitrec( digit input )
15 {
16   #include "training_data.h"
17
18   // This array stores K minimum distances per training set
19   bit6 knn_set[10][K_CONST];
20
21   // Initialize the knn set
22 DIGITREC_INIT_LOOP_OUTER:
23   for ( int i = 0; i < 10; ++i )
24 DIGITREC_INIT_LOOP_INNER:
25     for ( int k = 0; k < K_CONST; ++k )
26       // Note that the max distance is 49
27       knn_set[i][k] = DIGIT_SIZE+1;
28
29   for ( int i = 0; i < TRAINING_SIZE; ++i ) {
30 DIGITREC_PROC_LOOP_INNER:
31     for ( int j = 0; j < 10; j++ ) {
32
33       // Read a new instance from the training set
34       digit training_instance = training_data[j * TRAINING_SIZE + i];
35       // Update the KNN set
36       update_knn( input, training_instance, knn_set[j] );
37     }
38   }
```

◎圖 7 Bottleneck Of Code

```
  bit6 knn_set[10][K_CONST];
#pragma HLS ARRAY_PARTITION variable=knn_set complete dim=0
```

◎圖 8 Array Partition(分割成獨立的 Register)

Solution2:Pipeline

在更新鄰居的過程中，若沒有使用 Pipeline 將會使 Latency 大大增加

```
59 void update_knn( digit test_inst, digit train_inst, bit6 min_distances[K_CONST] )
60 {
61 #pragma HLS PIPELINE
```

◎圖 9 Pipeline

Solution3:Solition1+ Solition2

Performance Estimates

Timing

Clock		Baseline	solution1	solution2	solution3
ap_clk	Target	10.00 ns	10.00 ns	10.00 ns	10.00 ns
	Estimated	6.379 ns	6.412 ns	8.724 ns	8.724 ns

Latency

		Baseline	solution1	solution2	solution3
Latency (cycles)	min	993837	993837	417837	399837
	max	1101897	1029897	417897	399897
Latency (absolute)	min	9.938 ms	9.938 ms	4.178 ms	3.998 ms
	max	11.019 ms	10.299 ms	4.179 ms	3.999 ms
Interval (cycles)	min	993837	993837	417837	399837
	max	1101897	1029897	417897	399897

Utilization Estimates

	Baseline	solution1	solution2	solution3
BRAM_18K	96	96	100	100
DSP48E	0	0	0	0
FF	296	1039	696	5920
LUT	985	3394	3242	7247
URAM	0	0	0	0

◎圖 10 Compare Report(可以看出 Solution3 相對於 Baseline 在 Latency 有減少，尤其是 Pipeline 的幫助很大)

3. Github submit (1%)

https://github.com/r08943099/MSOCFall2020

4. Complexity (1% - Instructor/ TA decide)