ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδιασμός Ενσωματωμένων Συστημάτων

9ο Εξάμηνο, 2024-2025

4η Εργαστηριακή Άσκηση

αναφορά της φοιτήτριας:

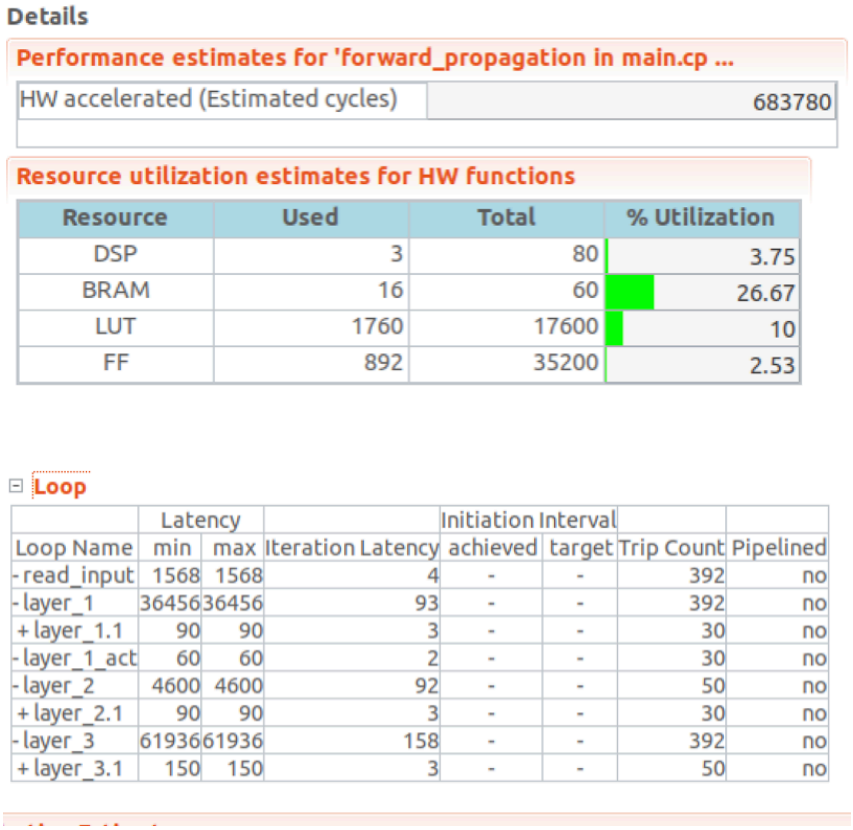
Λάζου Μαρία-Αργυρώ (el20129)

Ομάδα:**35**

ΑΣΚΗΣΗ 1. Performance & Resource Measurement

(A) Estimation performance στο unoptimized design

Μη έχοντας επιχειρήσει κάποια βελτιστοποίηση, οι εκτιμώμενοι κύκλοι εκτέλεσης της εφαρμογής καταγράφονται ίσοι με 683780 όπως φαίνεται στο στιγμιότυπο που ακολουθεί. Ακόμη παρατίθενται οι λεπτομέρειες από το HLS Report για τα Loops και βλέπουμε ότι ο compiler δεν έχει πραγματοποιήσει κανένα pipelined design αφού δεν δώσαμε άλλωστε κατάλληλα hints.



Details

Performance estimates for 'forward_propagation in main.cp ...

HW accelerated (Estimated cycles)	683780

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	3	80	3.75
BRAM	16	60	26.67
LUT	1760	17600	10
FF	892	35200	2.53

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- read_input	1568	1568	4	-	-	392	no
- layer_1	36456	36456	93	-	-	392	no
+ layer_1.1	90	90	3	-	-	30	no
- layer_1_act	60	60	2	-	-	30	no
- layer_2	4600	4600	92	-	-	50	no
+ layer_2.1	90	90	3	-	-	30	no
- layer_3	61936	61936	158	-	-	392	no
+ layer_3.1	150	150	3	-	-	50	no

(B) Hardware Evaluation στο unoptimized design

Ακολουθώντας τα βήματα παραγωγής του bitstream και εκτέλεσης στο Zybo, κατγράφεται η έξοδος όπως τυπώθηκε στο screen terminal. Συγκεκριμένα, οι κύκλοι που αποιτούνται για την hardware version του αλγορίθμου υπολογίστηκαν ίσοι με 682899 ενώ το speedup σε σχέση με την software version είναι 2.16059. Παρατηρώ ότι η προσομοίωση είναι αρκετά αντικειμενική στην εκτίμηση των κύκλων και μάλλον ελάχιστα πιο απαισιόδοξη το οποίο βολεύει για την ανάλυση μας έπειτα.

```
sh-4.3#  ./gan.elf
Setting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 682899
Software cycles : 1475467
Speed-up        : 2.16059
Saving results to output.txt...
sh-4.3#
```

(Γ) Design Space Exploration

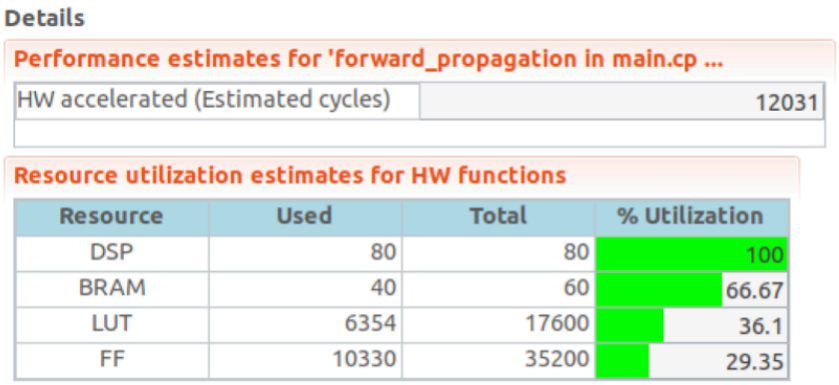
Τα optimisations που δοκιμάστηκαν εκμεταλλεύονται τις δυνατότητες για παραλληλισμό σε επίπεδο δεδομένων (data-level parallelism) και ταυτόχρονες προσβάσεις μνήμης. Παρατηρώ ότι όλα τα layers του νευρωνικού δικτύου δεν παρουσιάζουν εξαρτήσεις στο εσωτερικότερο loop οπότε τα iterations πάνω στην μεταβλητή j μπορούν να εκτελεστούν ταυτόχρονα. Με την τρέχουσα αναπαράσταση των διανυσμάτων βαρών W1, W2, W3 σε ένα εννοιαίο πίνακα για το καθένα, μπορούν να πραγματοποιηθούν το πολύ 2 προσβάσεις ανά κύκλο. Για τον λόγο αυτό, πραγματοποιείται διαμέριση σε υποπίνακες μεγέθους όσο το άνω όριο του εκάστοτε εσωτερικού loop ως προς τις στήλες για το W1 και ως προς τις γραμμές για τα W2, W3. Έτσι, σε 1 κύκλο επιτρέπουμε στην εφαρμοφή να διαβάσει όλα τα δεδομένα που απιτούνται. Παιρεταίρω κατάτμιση δεν έχει νόημα και θα αποτελούσε μονάχα σπατάληση πόρων. Για να επιτευχθεί το παραπάνω, ορίζεται ρητά unroll με παράγοντα ίσο με το εύρος των επαναλήψεων (ακριβώς ίδια αποτελέσματα λαμβάνονται και με την χρήση pipeline). Τέλος, ορίζω τις επιμέρους συναρτήσεις relu, tanh ως inline (αν και ο compiler τις έκανε έτσι και αλλιώς). Οι προσθήκες φαίνονται στο απόσπασμα κώδικα που ακολουθεί:

network.cpp

```
#include "network.h"
#include "weight_definitions.h"
#include "tanh.h"

l_quantized_type ReLU(l_quantized_type res)
{
#pragma HLS inline
  if (res < 0)
    return 0;

  return res;
}

l_quantized_type tanh(l_quantized_type res)
{
  #pragma HLS inline
  #pragma HLS pipeline II=1
  if (res >= 2)
    return 1;
  else if (res < -2)
    return -1;
  else
  {
    ap_int <BITS+2> i = res.range(); //prepare result to match tanh value
    return tanh_vals[(BITS_EXP/2) + i.to_int()];
  }

}

void forward_propagation(float *x, float *y)
{
  quantized_type xbuf[N1];
  l_quantized_type layer_1_out[M1];
  l_quantized_type layer_2_out[M2];

  #pragma HLS array_partition variable=W1 factor=15 dim=2
  #pragma HLS array_partition variable=layer_1_out factor=15
  #pragma HLS array_partition variable=xbuf factor=15

  #pragma HLS array_partition variable=W2 factor=15 dim=1
  #pragma HLS array_partition variable=layer_2_out factor=15

  #pragma HLS array_partition variable=W3 factor=25 dim=1


  //limit resources to max DSP number of Zybo - do not change
  #pragma HLS ALLOCATION instances=mul limit=80 operation

```

```
49    read_input:
50    for (int i=0; i<N1; i++)
51    {
52      #pragma HLS pipeline II=1
53      xbuf[i] = x[i];
54    }
55
56    // Layer 1
57    layer_1:
58    for(int i=0; i<N1; i++)
59    {
60      #pragma HLS pipeline II=1
61      for(int j=0; j<M1; j++)
62      {
63        #pragma HLS unroll factor=30
64        l_quantized_type last = (i==0) ? (l_quantized_type) 0 : layer_1_out[j];
65        quantized_type term = xbuf[i] * W1[i][j];
66        layer_1_out[j] = last + term;
67      }
68    }
69
70    layer_1_act:
71    for(int i=0; i<M1; i++)
72    {
73      #pragma HLS pipeline II=1
74      layer_1_out[i] = ReLU(layer_1_out[i]);
75    }
76
77    layer_2:
78    for (int i = 0; i < M2; i ++)
79    {
80      l_quantized_type result_0 = 0;
81      #pragma HLS pipeline II=1
82      for (int j = 0; j < N2; j++)
83      {
84        #pragma HLS unroll factor=30
85        l_quantized_type term_0 = layer_1_out[j] * W2[j][i];
86
87        result_0 += term_0;
88      }
89      layer_2_out[i] = ReLU(result_0);
90    }
91
92    // Layer 3
93    layer_3:
94    for (int i = 0; i < M3; i ++)
95    {
96      l_quantized_type result = 0;
97      #pragma HLS pipeline II=1
98      for (int j = 0; j < N3; j++)
99      {
100       #pragma HLS unroll factor=50
101       l_quantized_type term_0 = layer_2_out[j] * W3[j][i];
102
103       result += term_0;
104     }
105     y[i] = tanh(result).to_float();
106   }
107 }
```

Οι συνολικοί κύκλοι που απαιτούνται με βάση την προσομοίωση για το optimized design είναι ίσοι με 12031 όπως καταγράφηκαν από το SDSoC:



Details

Performance estimates for 'forward_propagation in main.cp ...

HW accelerated (Estimated cycles)	12031

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	40	60	66.67
LUT	6354	17600	36.1
FF	10330	35200	29.35

Ακολουθώντας ξανά τα βήματα για την εξαγωγή του bitstream για το optimized code και loading στην sd card του Zybo, οι actual κύκλοι υπολογίστηκαν ίσοι με 12207 ενώ το speedup ανέρχεται σε 120.773. Τα αποτελέσματα απεικονίζονται παρακάτω όπως τυπώθηκαν στο screen terminal :

```
sh-4.3#  ./gan.elf
Setting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 12207
Software cycles : 1475467
Speed-up        : 120.773
Saving results to output.txt...
sh-4.3#
```

Επιπλέον βελτιώσεις

Για τα layers 2 και 3 δεν υπάρχουν εξαρτήσεις μεταξύ των i επαναλήψεων οπότε η εκτέλεση μοιράζεται σε 2 μέρη που μπορούν να υπολογιστούν ταυτόχρονα. Κανονικά απαιτείται και διπλασιασμός του partitioning ως προς την άλλη διάσταση όμως υπερβαίνει τους διαθέσιμους πόρους του Zybo, οπότε επωφελούμαι μόνο από τα overheads των branches στα loops. Ο ανανεωμένος κώδικας ακολουθεί:

network2.cpp

```
#include "network.h"
#include "weight_definitions.h"
#include "tanh.h"

l_quantized_type ReLU(l_quantized_type res)
{
  #pragma HLS inline
  if (res < 0)
  return 0;

  return res;
}

l_quantized_type tanh(l_quantized_type res)
{
#pragma HLS inline
  if (res >= 2)
    return 1;
  else if (res < -2)
    return -1;
  else
  {
    ap_int <BITS+2> i = res.range(); //prepare result to match tanh value
    return tanh_vals[(BITS_EXP/2) + i.to_int()];
```

```
  }
}

void forward_propagation(float *x, float *y)
{
  quantized_type xbuf[N1];
  l_quantized_type layer_1_out[M1];
  l_quantized_type layer_2_out[M2];


  #pragma HLS array_partition variable=W1 factor=30 dim=2
  #pragma HLS array_partition variable=layer_1_out factor=30
  #pragma HLS array_partition variable=xbuf factor=30

  #pragma HLS array_partition variable=W2 factor=30 dim=1
  #pragma HLS array_partition variable=layer_2_out factor=30

  #pragma HLS array_partition variable=W3 factor=50 dim=1

  //limit resources to max DSP number of Zybo - do not change
  #pragma HLS ALLOCATION instances=mul limit=80 operation

  read_input:
  for (int i=0; i<N1; i++)
  {
    #pragma HLS pipeline II=1
    xbuf[i] = x[i];
  }

// Layer 1
  layer_1:
  for(int i=0; i<N1; i++)
  {
    #pragma HLS pipeline II=1
    for(int j=0; j<M1; j++)
    {
      #pragma HLS unroll factor=30
      l_quantized_type last = (i==0) ? (l_quantized_type) 0 : layer_1_out[j];
      quantized_type term = xbuf[i] * W1[i][j];
      layer_1_out[j] = last + term;
    }
  }

  layer_1_act:
    for(int i=0; i<M1; i++)
    {
      #pragma HLS pipeline II=1
      layer_1_out[i] = ReLU(layer_1_out[i]);
    }

// Layer 2
  layer_2:
  for(int i=0; i<M2/2; i++)
  {
    #pragma HLS pipeline II=1
    int ii = M2/2 + i;
    l_quantized_type resulti = 0, resultii = 0;
    for(int j=0; j<N2; j++)
    {
      #pragma HLS unroll factor=30
      l_quantized_type termi = layer_1_out[j] * W2[j][i];
      l_quantized_type termii = layer_1_out[j] * W2[j][ii];
      resulti += termi;
      resultii += termii;
    }
    layer_2_out[i] = ReLU(resulti);
    layer_2_out[ii] = ReLU(resultii);
  }

// Layer 3
  layer_3:
  for(int i=0; i<M3/2; i++)
  {
    #pragma HLS pipeline II=1
    int ii = M3/2 + i;
```

```
100       l_quantized_type resulti = 0, resultii = 0;
101       for(int j=0; j<N3; j++)
102       {
103         #pragma HLS unroll factor=50
104         l_quantized_type termi = layer_2_out[j] * W3[j][i];
105         l_quantized_type termii = layer_2_out[j] * W3[j][ii];
106         resulti += termi;
107         resultii +=termii;
108       }
109       y[i] = tanh(resulti).to_float();
110       y[ii] = tanh(resultii).to_float();
111     }
112   }
```

Ο νέος εκτιμώμενος χρόνος σε κύκλους καταγράφηκε ίσος με 11875.

Details

Performance estimates for 'forward_propagation in main.cp ...

HW accelerated (Estimated cycles)	11875

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	34	60	56.67
LUT	9903	17600	56.27
FF	11594	35200	32.94

Οι αντίστοιχοι κύκλοι που υπολογίστηκαν στο Zybo :

```
sh-4.3#  ./gan.elf
Setting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 11779
Software cycles : 1475467
Speed-up        : 125.262
Saving results to output.txt...
sh-4.3#
```

(Δ) HLS Resource Profile

Για την αρχική υλοποίηση τα loop latency details φαίνονται παρακάτω:

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- read_input	394	394	4	1	1	392	yes
- layer_1	393	393	3	1	1	392	yes
- layer_1_act	30	30	1	1	1	30	yes
- layer_2	52	52	4	1	1	50	yes
- layer_3	400	400	10	1	1	392	yes

Έχει επιτευχθεί πλήρως pipelined design όμως στο layer 3, το οποίο είναι και το bottleneck, το Iteration latency είναι 10 και πιθανά οφείλεται στην πολυπλοκότητα της tanh συγκριτικά με την relu και στο πέρασμα του αποτελέσματος στο output y το οπόιο δεν μπορεί να παραληλοποιηθεί περαιτέρω αφού η μεταφορά γίνεται ως byte streams.

Για την δεύτερη υλοποίηση τα αντίστοιχα αποτελέσματα είναι :

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- read_input	394	394	4	1	1	392	yes
- layer_1	393	393	3	1	1	392	yes
- layer_1_act	30	30	1	1	1	30	yes
- layer_2	28	28	4	1	1	25	yes
- layer_3	400	400	11	2	1	196	yes

Παρατηρούμε ότι στο layer 3 δεν επιτεύχθηκε πλήρως pipeline, ωστόσο η επίδοση είναι συνολικά καλύτερη.

Τέλος, στο **Expression** section κατγράφονται όλοι οι καταχωρητές που χρησιμοποιούνται για πολλαπλασιασμούς, προσθέσεις και logical operations. Στην περίπτωση των πρώτων, δεν χρειάζονται καθόλου DSPs και εκτελούνται αποκλειστικά σε LUTs, ενώ στην περίπτωση των πολλαπλασιασμών χρειάζεται 1 DSP ανά πράξη. Η εφαρμογή δίνει όλα τα DSPs (80 συνολικά) σε πολλαπλασιασμούς καθώς είναι η πιο computational intensive πράξη στον συγκεκριμένο αλγόριθμο και με τον τρόπο αυτό εκτελείται σε 1 κύκλο, αφήνοντας ελεύθερα LUTs και FFs για άλλα computations. Ενδεικτικά απεικονίζονται κάποιες διαφορετικές καταγραφές ανάθεσης πόρων :

Variable Name	Operation	DSP48	F	LU	Bitwidth P0	Bitwidth P1
grp_fu_4767_p2	*	1	0	0	17	10
grp_fu_4768_p2	*	1	0	0	17	17
layer_1_out_23_V_1_fu_25314_p2	+	0	0	17	17	17
sel_tmp7_fu_20116_p2	and	0	0	1	1	1
tmp_12_fu_19995_p2	ashr	0	0	161	54	54
exitcond1_fu_19856_p2	icmp	0	0	3	9	8
F2_fu_19930_p2	-	0	0	12	11	12
sel_tmp435_demorgan_fu_20105_p2	or	0	0	1	1	1
layer_1_out_0_V_4_fu_26004_p3	select	0	0	16	1	1

ΑΣΚΗΣΗ 2. Quality Measurement

(A) Οι combined εικόνες για τα ζητούμενα indices (10,11,12) που παράχθηκαν από το software και το hardware αντίστοιχα απεικονίζονται στην συνέχεια :

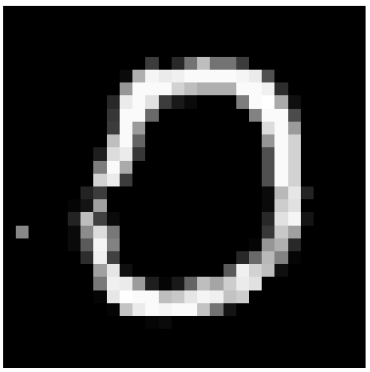


Figure 1: SW, idx = 10

,

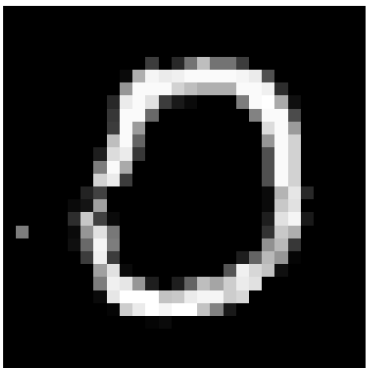


Figure 4: HW, idx = 10

,

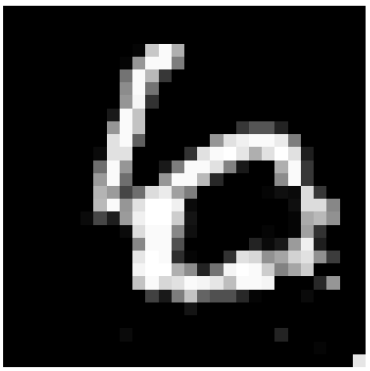


Figure 2: SW, idx = 11

,

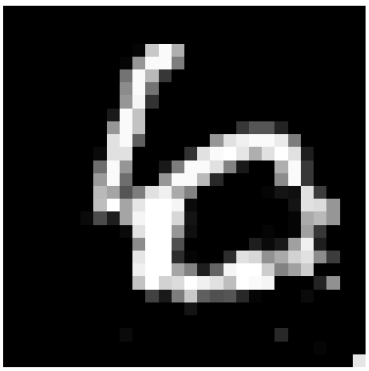


Figure 5: HW, idx = 11

,

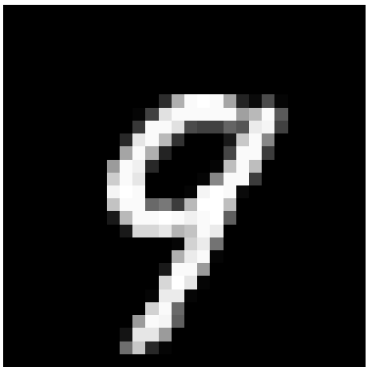


Figure 3: SW, idx = 12

,

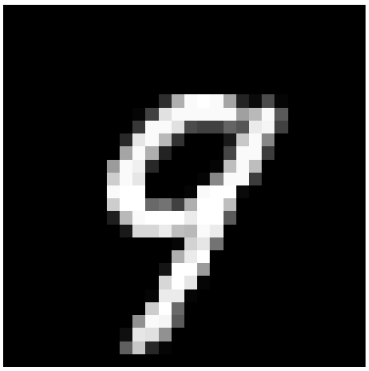


Figure 6: HW, idx = 12

,

(B) Για τον νέο υπλογισμό των τιμών της εφαπτομένης στο εύρος [−2,2] και για την ακρίβεια στα σημεία που προκύπτουν από την κβάντιση του άνω διαστήματος με βήμα ίσο με *BITS_EXP*, χρησιμοποιήθηκε το ακόλουθο python script που παράγει αυτόματα όλα τα header files:

tanh.py

```python
import numpy as np

def precompute_tanh(bits) :
    start = -2.0
    end = 2.0
    factor = 2**(bits+2)
    step = (end-start)/ factor

    x_vals = np.arange(start, end, step)
    tanh_vals = np.tanh(x_vals)

    with open(f"tanh_{bits}.h", 'w') as hpp:
            hpp.write(f"// Mapping tanh values from -2 to {(end-step):.16f} with a step of {step:.16f}\n")
        hpp.write(f"quantized_type tanh_vals[{factor}] = {{ ")
        for i, val in enumerate(tanh_vals):
            hpp.write(f"{val:.16f}, ")
        hpp.write("};\n")

if __name__ == '__main__' :
    for bits in [3,4,5,6,7,8,9,10]:
        precompute_tanh(bits)
```

Στον αρχικό κώδικα προστίθενται τα κατάλληλα definitions :

network_bits.cpp

```cpp
#include "network.h"
#include "weight_definitions.h"
// #include "tanh.h"

#if BITS==3
#include "tanh_3.h"
#elif BITS==4
#include "tanh_4.h"
#elif BITS==5
#include "tanh_5.h"
#elif BITS==6
#include "tanh_6.h"
#elif BITS==7
#include "tanh_7.h"
#elif BITS==8
#include "tanh_8.h"
#elif BITS==9
#include "tanh_9.h"
#elif BITS==10
#include "tanh_10.h"
#endif

l_quantized_type ReLU(l_quantized_type res)
{
#pragma HLS inline
  if (res < 0)
    return 0;

  return res;
}

l_quantized_type tanh(l_quantized_type res)
{
#pragma HLS inline
#pragma HLS pipeline II=1
  if (res >= 2)
```

```
    return 1;
  else if (res < -2)
    return -1;
  else
  {
    ap_int <BITS+2> i = res.range();      //prepare result to match tanh value
    return tanh_vals[(BITS_EXP/2) + i.to_int()];
  }
}

void forward_propagation(float *x, float *y)
{
  quantized_type xbuf[N1];
  l_quantized_type layer_1_out[M1];
  l_quantized_type layer_2_out[M2];

  #pragma HLS array_partition variable=W1 factor=15 dim=2
  #pragma HLS array_partition variable=layer_1_out factor=15
  #pragma HLS array_partition variable=xbuf factor=15

  #pragma HLS array_partition variable=W2 factor=15 dim=1
  #pragma HLS array_partition variable=layer_2_out factor=15

  #pragma HLS array_partition variable=W3 factor=25 dim=1


  //limit resources to max DSP number of Zybo - do not change
  #pragma HLS ALLOCATION instances=mul limit=80 operation

  read_input:
  for (int i=0; i<N1; i++)
  {
  #pragma HLS pipeline II=1
    xbuf[i] = x[i];
  }

  // Layer 1
  layer_1:
  for(int i=0; i<N1; i++)
  {
    #pragma HLS pipeline II=1
    for(int j=0; j<M1; j++)
    {
      #pragma HLS unroll factor=30
      l_quantized_type last = (i==0) ? (l_quantized_type) 0 : layer_1_out[j];
      quantized_type term = xbuf[i] * W1[i][j];
      layer_1_out[j] = last + term;
    }
  }

  layer_1_act:
  for(int i=0; i<M1; i++)
  {
    #pragma HLS pipeline II=1
    layer_1_out[i] = ReLU(layer_1_out[i]);
  }

  layer_2:
  for (int i = 0; i < M2; i ++)
  {
    l_quantized_type result_0 = 0;
    #pragma HLS pipeline II=1
    for (int j = 0; j < N2; j++)
    {
      #pragma HLS unroll factor=30
      l_quantized_type term_0 = layer_1_out[j] * W2[j][i];

      result_0 += term_0;
    }
    layer_2_out[i] = ReLU(result_0);
  }

  // Layer 3
  layer_3:
  for (int i = 0; i < M3; i ++)
```

```
    {
      l_quantized_type result = 0;
      #pragma HLS pipeline II=1
      for (int j = 0; j < N3; j++)
      {
        #pragma HLS unroll factor=50
        l_quantized_type term_0 = layer_2_out[j] * W3[j][i];

        result += term_0;
      }
      y[i] = tanh(result).to_float();
    }
  }
```

Τέλος, για την συγρκιτική ανάλυση των τιμών μόνο του παραγόμενου αρχείου εξόδου και εφόσον ο ακριβής προσδιορισμός των κύκλων έχει μετρηθεί στην άσκηση 1 πάνω στο Zybo, χρησιμοποιήθηκε το ακόλουθο script για την μεταγλώττιση του πηγαίου κώδικα της αφρμογής και προσομοίωση του hardware υπολογισμού στον τοπικό μου υπολογιστή:

```
#!/bin/bash
for bits in 3 4 5 6 7 8 9 10; do
    echo "Compiling main.cpp for BITS=$bits"
     g++ -DBITS=$bits -O2 -I/opt/Xilinx/SDx/2016.4/Vivado_HLS/include -ITanh/ main.cpp network.cpp -
o a.out

    ./a.out data.txt
    mv output.txt "output_bits${bits}.txt"
    echo "Output for BITS=$bits saved to output_bits${bits}.txt"
done
```

Οι εικόνες που προκύπτουν από την επεξεργασία των αρχείων εξόδου στο jupyter notebook που μας δόθηκε απεικονίζονται παρακάτω:

BITS = 3

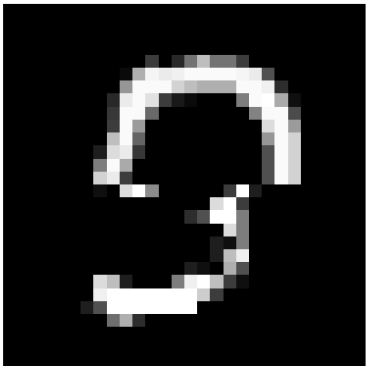


Figure 7: HW, idx = 10

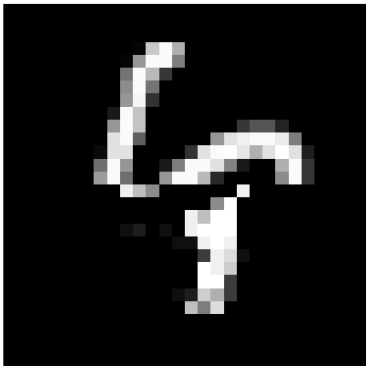


Figure 8: HW, idx = 11

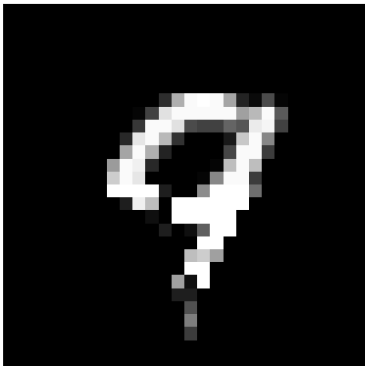


Figure 9: HW, idx = 12

BITS = 4

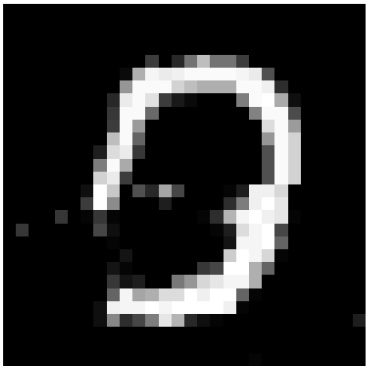


Figure 10: HW, idx = 10

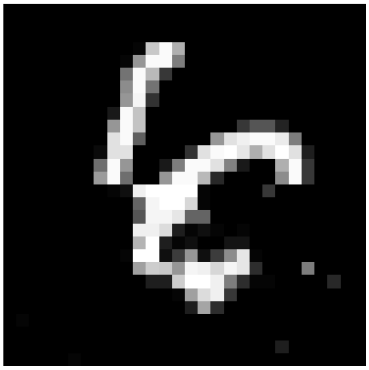


Figure 11: HW, idx = 11

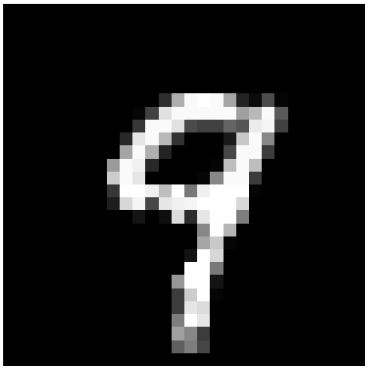


Figure 12: HW, idx = 12

BITS = 5

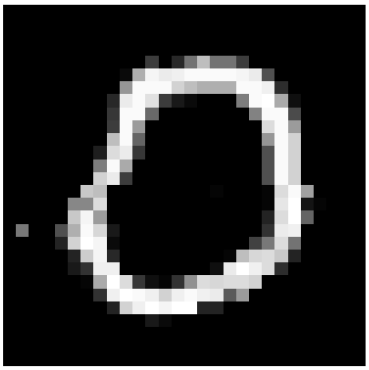


Figure 13: HW, idx = 10

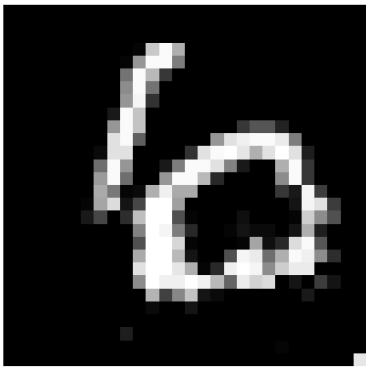


Figure 14: HW, idx = 11

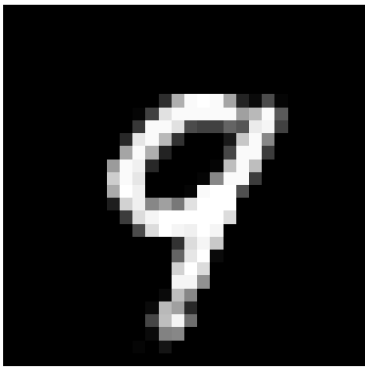


Figure 15: HW, idx = 12

BITS = 6

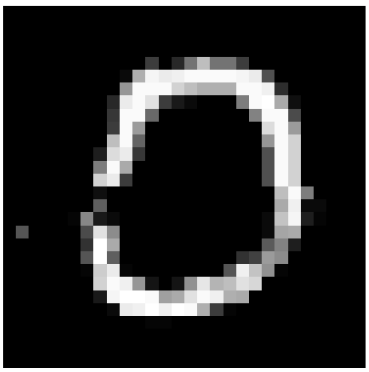


Figure 16: HW, idx = 10

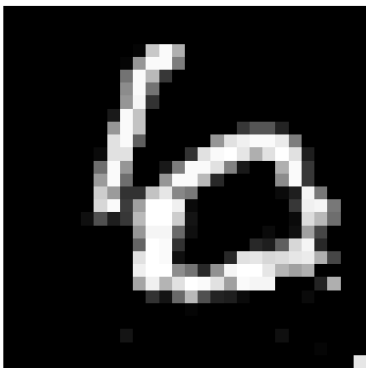


Figure 17: HW, idx = 11

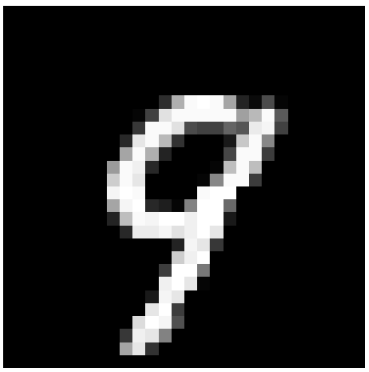


Figure 18: HW, idx = 12

BITS = 7

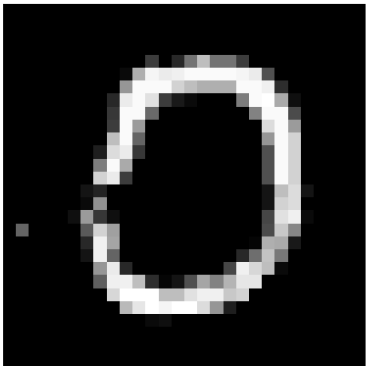


Figure 19: HW, idx = 10

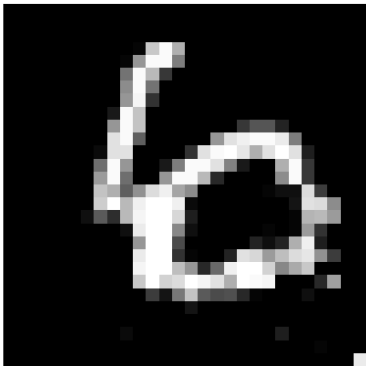


Figure 20: HW, idx = 11

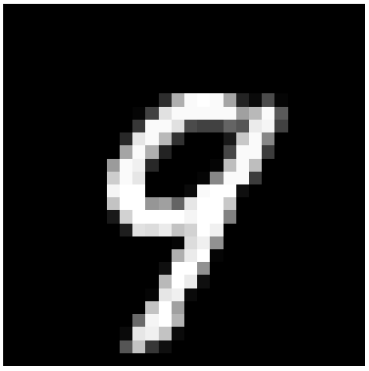


Figure 21: HW, idx = 12

BITS = 8

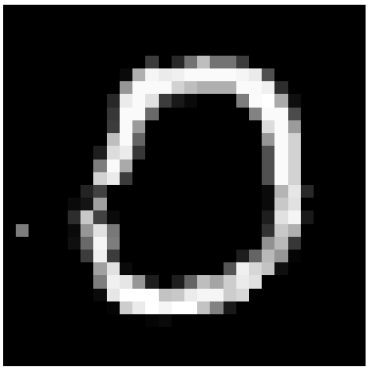


Figure 22: HW, idx = 10

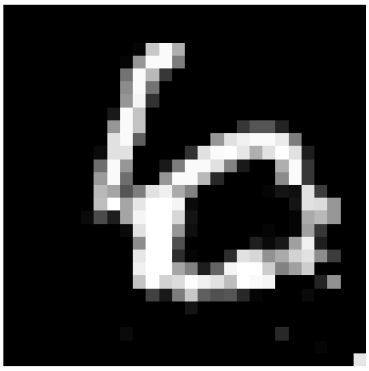


Figure 23: HW, idx = 11

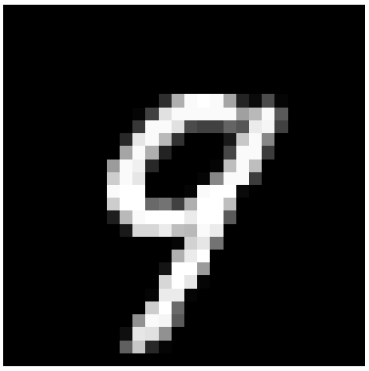


Figure 24: HW, idx = 12

BITS = 9

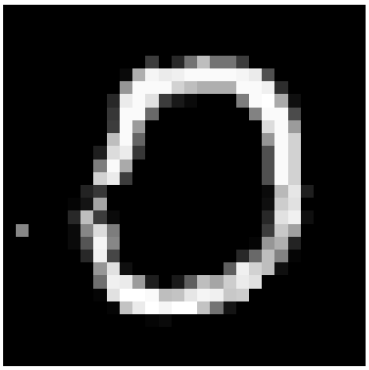


Figure 25: HW, idx = 10

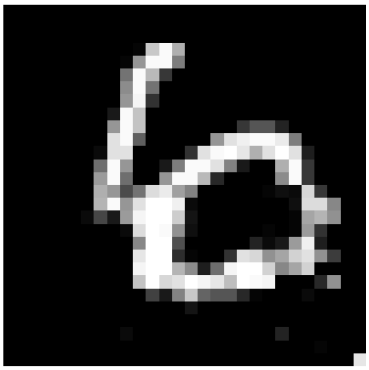


Figure 26: HW, idx = 11

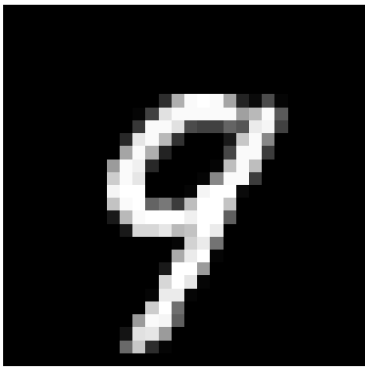


Figure 27: HW, idx = 12

BITS = 10

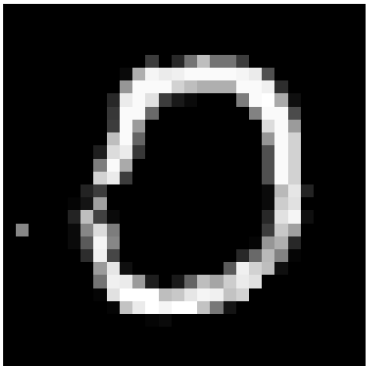


Figure 28: HW, idx = 10

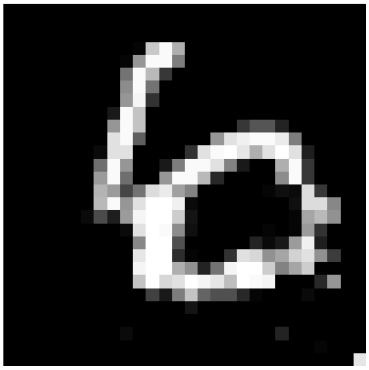


Figure 29: HW, idx = 11

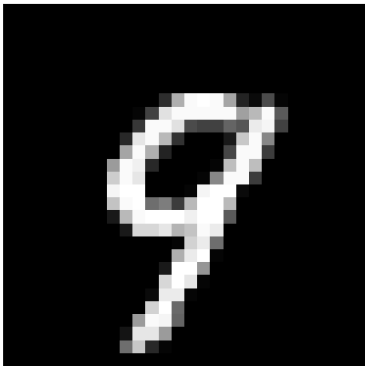


Figure 30: HW, idx = 12

Οι εικόνες που παράγονται από το software είναι πανομοιότυπες όπως είναι αναμενόμενο, εφόσον η ακρίβεια των floating point αριθμών της αρχιτεκτονικής του μηχανήματος είναι fixed και ο κώδικας δεν μεταβάλλεται. Γι' αυτό, άλλωστε, χρησιμοποιούνται έπειτα ως σημείο αναφοράς των σφαλμάτων στο επόμενο ερώτημα.
Παρατηρώ ότι καθώς αυξάνονται τα BITS, η εικόνες έχουν καλύτερη ποιότητα για όλα τα indices με ένα άνω φράγμα στα 7. Συγκεκριμένα, για τις περιπτώσεις BITS = 4, ο θόρυβος είναι μεγάλος και τα περιθώρια των αριθμών δεν είναι καθόλου σαφή, ενώ για BITS = 10 τα νούμερα διακρίνονται πλήρως από το background και έχουν σαφή περιθώρια.
Αυτό οφείλεται στον διαφορετικό παράγοντα κβάντισης σε κάθε περίπτωση. Μικρότερο step (δηλ. μεγαλύτερο quantization) οδηγεί σε πιο λεπτομερή απεικόνιση των τιμών εισόδων και επηρεάζει την ευαισθησία της συνάρτησης ενεργοποίησης στο layer 3 που είναι μάλιστα και το εξωτερικό επίπεδο του νευρωνικού. Οδηγεί, λοιπόν, σε πιο ακριβείς προβλέψεις.

(Γ) Οι μετρικές που χρησιμοποιήθηκαν για την ανάλυση της ποιότητας ανακατσκευής των εικόνων είναι η Maximum Pixel Error που δίνεται από τον τύπο:

$$\text{Max Error} = \max\left(|\ I_{\text{sw}(i,j)} - I_{\text{hw}(i,j)}|\right)$$

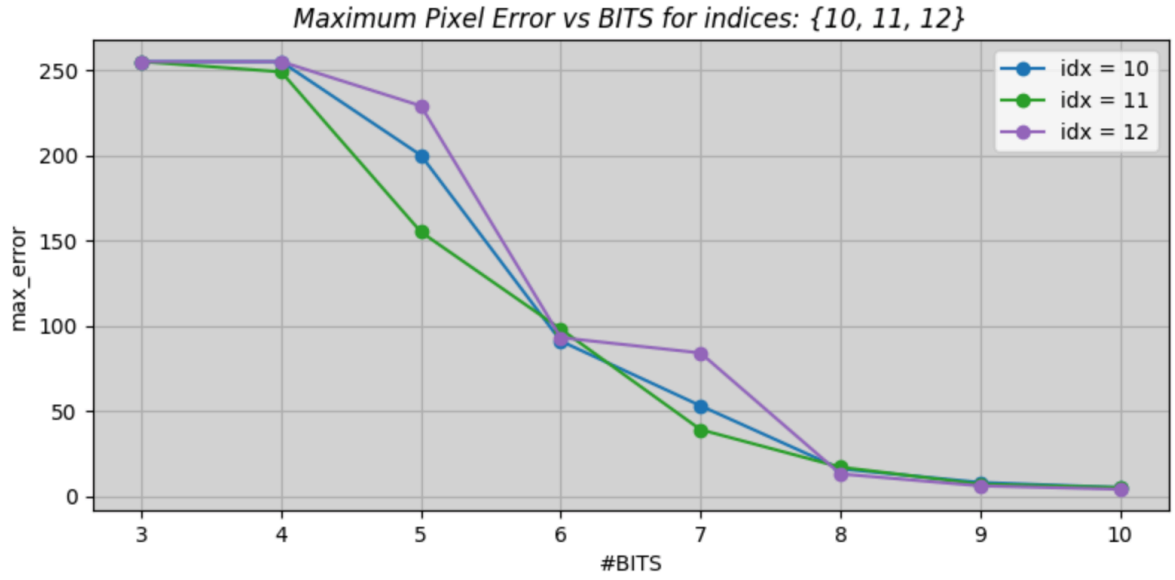
και η Peak Signal-to-Noise Ratio που δίνεται από τον τύπο :

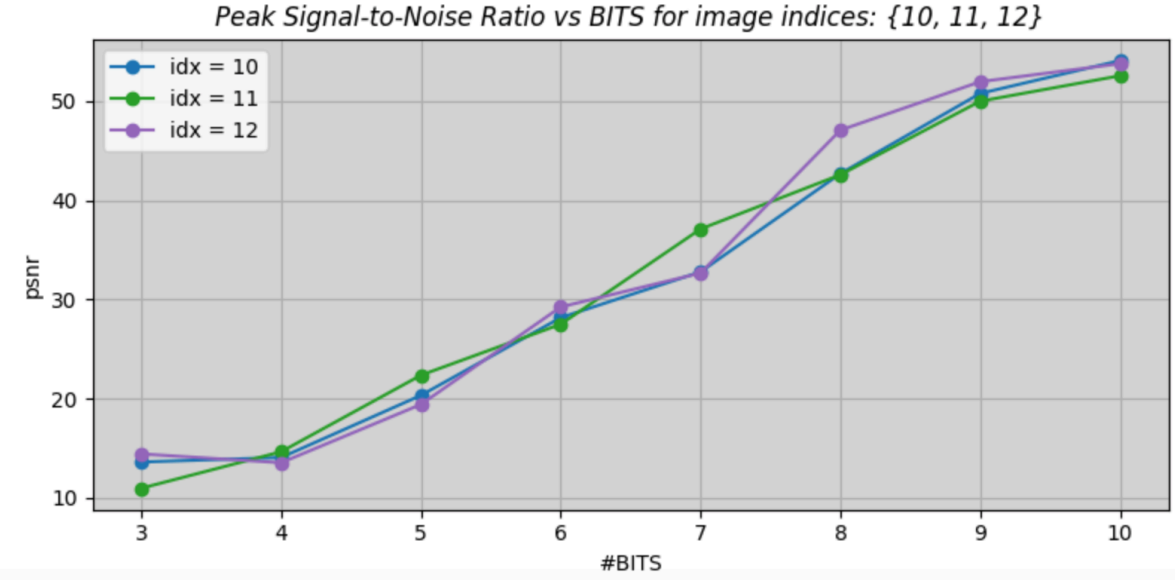
$$\text{PSNR} = 10\log_{\{10\}}\left(\frac{\text{MAX}_I^2}{\text{MSE}}\right)$$

, όπου

$$\text{MSE} = \frac{1}{\text{MN}}\sum_{i=1}^{M}\sum_{j=1}^{N}\left(I_{\text{sw}(i,j)} - I_{\text{hw}(i,j)}\right)^2$$

Οι γραφικές παραστάσεις και ο πίνακας που ακολουθούν, συνοψίζουν τις τιμές τους όπως εκτιμήθηκαν από τον επιμέρους κώδικα του jupyter notebook που μας δόθηκε, για όλους τους πιθανούς συνδυασμούς *(BITS, idx)*.





BITS	index	Maximum Pixel Error	Peak Signal-to-Noise Ratio
3	10	255	13.598006427897612
3	11	255	10.951273518816551
3	12	255	14.412458999930394
4	10	255	14.051663945384881
4	11	249	14.634988266184102
4	12	255	13.525831164368576
5	10	200	20.32370043626041
5	11	155	22.354010215775265
5	12	229	19.407263480245117
6	10	91	28.159872103979914
6	11	98	27.496646597732962
6	12	93	29.23557136050292
7	10	53	32.77870229255662
7	11	39	37.09878984137624
7	12	84	32.654500509492976
8	10	16	42.6822168370888
8	11	17	42.56993337396983
8	12	13	47.065287020211215
9	10	8	50.76968548527325
9	11	7	49.98975523417636
9	12	6	51.955130625734746
10	10	5	54.08543347142643
10	11	5	52.556099880280584
10	12	4	53.76982650203158

Table 4: Αναλυτικός πίνακας μετρικών αξιολόγησης

Το Max Pixel Error δείχνει την χειρότερη απόκλιση ενός μεμονωμένου σημείου, άρα παρέχει κάποια πληροφορία σχετικά με το μέγεθος της διαστρεύλωσης που μπορεί να έχει η εικόνα σε ακραίες περιπτώσεις. Το PSNR υπολογίζεται με βάση τον μέσο όρο σφαλμάτων των pixels, είναι λογαριθμικό επομένως εκφράζει μια πιο σφαιρική εκτίμηση του θορύβου πάνω στην εικόνα. Παρ'όλα αυτά δεν λαμβάνει υπ'οψιν τον βιολογικό παράγοντα του ανθρώπινου ματιού, γι'αυτό ενώ οι καμπύλες προκύποτυν γνησίως αύξουσες, δεν μπορούμε να διακρίνουμε ανάλογη βελτίωση στις εικόνες από τα 6 BITS και πάνω. Αντιθέτως, οι καμπύλες του Max Error φτάνουν ικανοποιητικό σημείο σύγκλισης στα 8 BITS και είναι πιο σταθερές από εκεί και πάνω. Με βάση τα οπτικά αποτελέσματα αποτελέσματα, η πρώτη μετρική είναι πιο αντιπροσωπευτική. Προσωπικά, δεν διακρίνω διαφορά ούτε με τα 7 BITS.

Δεδομένου ότι η αναπαράσταση των pixels με περισσότερα bits κοστίζει σε απαιτήσεις μνήμης και η βέλτιστη ποιότητα της εικόνας στα πλάισια που συλλαμβάνει το ανθρώπινο μάτι πετυχαίνεται με την χρήση 7 bits κατά την γνώμη μου, θα το θεωρούσα ιδανική επιλογή. Το νευρωνικό δίκτυο δεν επωφελείται από περισσότερη ακρίβεια και έχει ήδη εκαπιδευτεί να διαχωρίζει τα patterns.

Παράρτημα

Ο κώδικας για την εξαγωγή των γραφικών παραστάσεων μέσα στο τροποποιημένο *plot_outputs.ipynb* δίνεται παρακάτω:

plots.py

```
import csv

resfile = os.path.join(img_dir, "metrics.csv")

def save_results():
  with open(resfile, mode='w', newline='') as file:
      writer = csv.writer(file)
      writer.writerow(['BITS', 'index', 'Maximum Pixel Error', 'Peak Signal-to_Noise Ratio'])

      for key in max_err_vals:
          b, i = key
          max_err = max_err_vals[key]
          psnr_val = psnr_vals[key]
          writer.writerow([b, i, max_err, psnr_val])

def plot_results():
  fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 8))
  ax1.set_facecolor("#d3d3d3")
  ax2.set_facecolor("#d3d3d3")
  colors =  [
    "#1f77b4",  # Blue
    "#2ca02c",  # Green
    "#9467bd"  #Purple
  ]

  # Plot max_err values
  for i in idx:
      ax1.plot(bits, [max_err_vals[(b, i)] for b in bits], color = colors[i-10], marker='o',
label=f'idx = {i}')
  ax1.grid()
  ax1.set_xlabel("#BITS")
  ax1.set_ylabel("max_error")
       ax1.set_title("Maximum Pixel Error vs BITS for indices: {10, 11,
12}",fontsize=12,fontstyle='italic')
  ax1.legend()

  # Plot psnr values
  for i in idx:
       ax2.plot(bits, [psnr_vals[(b, i)] for b in bits], color = colors[i-10], marker='o',
label=f'idx = {i}')
  ax2.grid()
  ax2.set_xlabel("#BITS")
  ax2.set_ylabel("psnr")
    ax2.set_title("Peak Signal-to-Noise Ratio vs BITS for image indices: {10, 11,
12}",fontsize=12,fontstyle='italic')
  ax2.legend()

  plt.tight_layout()
  plt.show()
  plt.savefig(os.path.join(img_dir, "metric_plots.png"))
```