Δημοκρίτειο Πανεπιστήμιο Θράκης

Τμήμα: Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εργαστήριο: Αρχιτεκτονικής Υπολογιστών και Συστημάτων Υψηλών Επιδόσεων

Εργασία για το μάθημα Παράλληλοι Αλγόριθμοι και Υπολογιστική Πολυπλοκότητα

Θέμα: **Εφαρμογή συνελικτικών φίλτρων και maxpooling σε RGB εικόνες**

Έτος: 2023-2024

Ομάδα 8

Ονοματεπώνυμο:

Γιάμπαστος Πουλέκας Δημήτριος ee: 58147

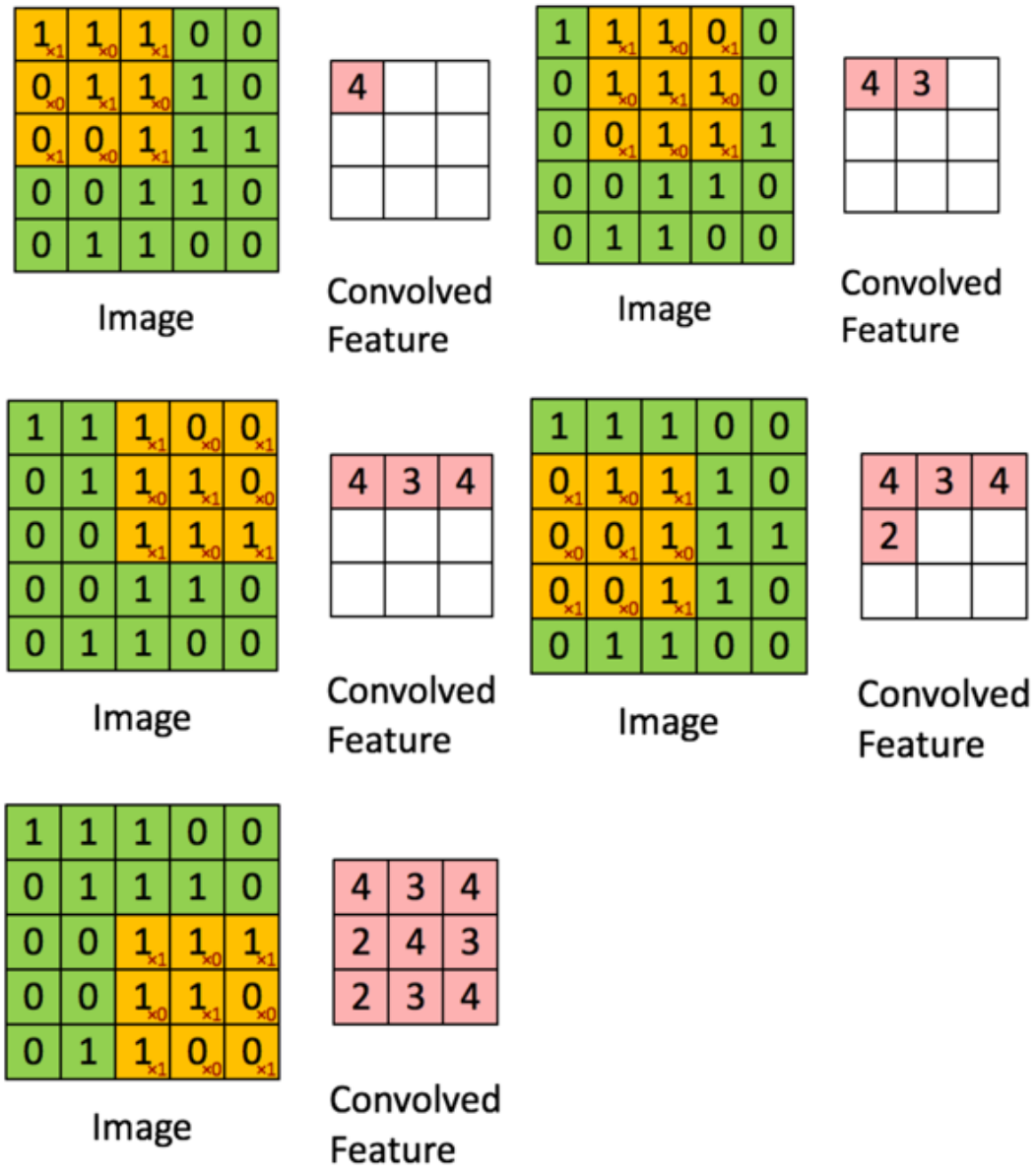
Μωραΐτη Παναγιώτα ee: 58054

Θεωρητική παρουσίαση του σειριακού αλγόριθμου

Η συνέλιξη (convolution) σε μια εικόνα είναι μια βασική πράξη που χρησιμοποιείται κυρίως σε τεχνικές επεξεργασίας εικόνας. Κατά τη συνέλιξη, το φίλτρο (πυρήνας-kernel) κινείται πάνω στην εικόνα και πραγματοποιεί πολλαπλασιασμούς των τιμών του με τις αντίστοιχες τιμές των pixel της εικόνας.

Η συνέλιξη βοηθά στην ανίχνευση χαρακτηριστικών στην εικόνα, όπως ακμές, σχήματα και υποδιαιρέσεις, επιτρέποντας την ανάλυση και την εξαγωγή σημαντικών πληροφοριών.

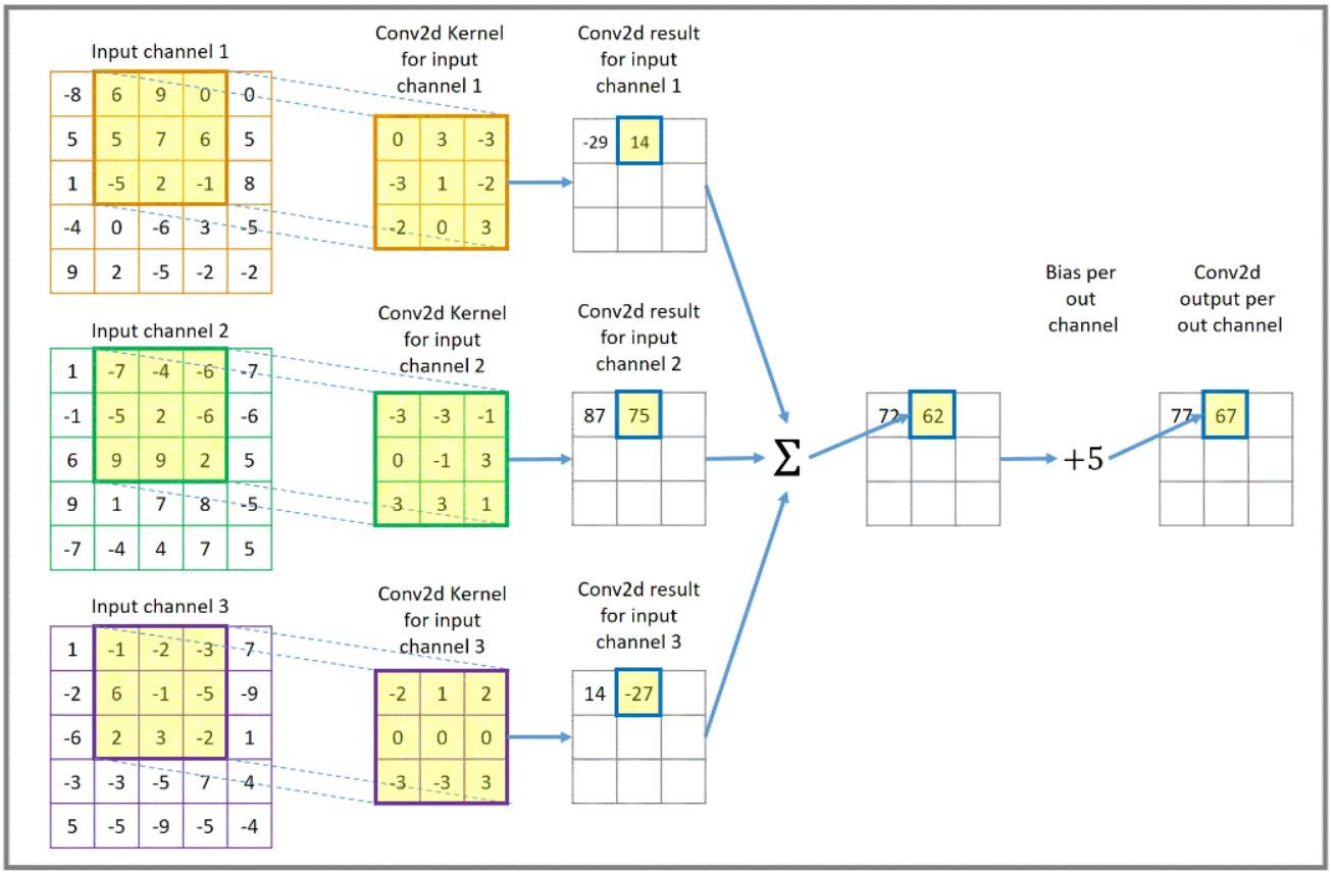
Στην παρακάτω εικόνα φαίνεται η συνέλιξη σε έναν πίνακα 5x5 με ένα kernel 3x3. Το αποτέλεσμα είναι ένας πίνακας 3x3 που ονομάζεται χάρτης χαρακτηριστικών (feature map).



Όταν εφαρμόζεται συνέλιξη σε μια εικόνα με τρία κανάλια RGB, χρησιμοποιούνται τρία φίλτρα. Η διαδικασία συνέλιξης λαμβάνει χώρα ξεχωριστά για κάθε κανάλι. Κάθε φίλτρο εφαρμόζεται στον αντίστοιχο κανάλι της εικόνας, πραγματοποιώντας συνέλιξη και δημιουργώντας έναν χάρτη χαρακτηριστικών (feature map).

Τα αποτελέσματα από τις τρεις συνελίξεις συνδυάζονται (αθροίζονται στοιχείο προς στοιχείο-Element-wise Addition), οπότε το τελικό αποτέλεσμα περιλαμβάνει πληροφορίες από όλα τα κανάλια της αρχικής εικόνας. Αυτή η διαδικασία επιτρέπει στο μοντέλο να αντλήσει σημαντικά χαρακτηριστικά από κάθε χρωματικό κανάλι και να ενσωματώσει τις πληροφορίες αυτές στην τελική αναπαράσταση. Επίσης, σε κάθε pixel του τελικού feature map, προστίθεται και μια σταθερά που ονομάζεται bias.

Στην παρακάτω εικόνα φαίνεται η συνέλιξη για τρία κανάλια. Η πράξη αυτή ονομάζεται convolution 2D, επειδή από μια εικόνα 3D (height x width x depth), μας πηγαίνει σε μια εικόνα 2D (height x width).



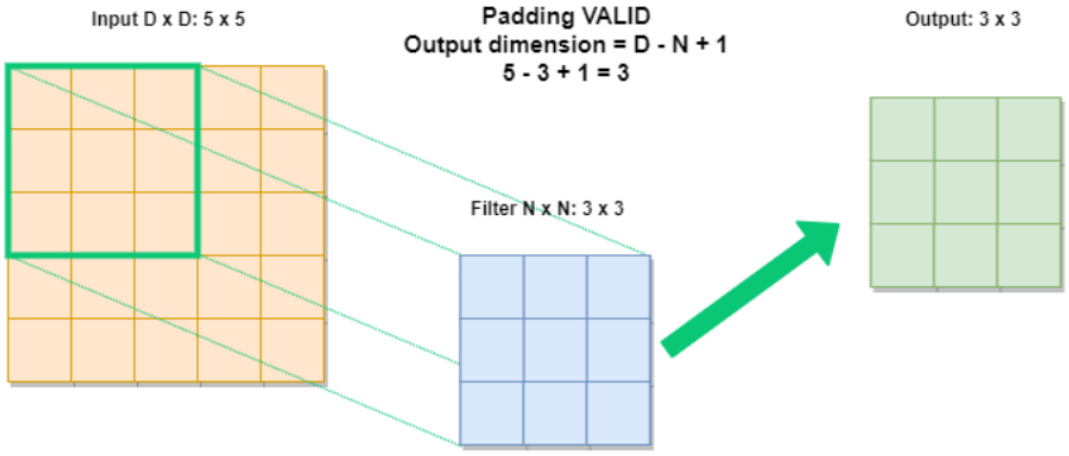
Κατά τη συνέλιξη, το φίλτρο κινείται πάνω στην εικόνα. Ωστόσο, όταν φτάνει στα άκρα της εικόνας, μεγάλο μέρος του φίλτρου βγαίνει εκτός της εικόνας. Αυτό μπορεί να οδηγήσει σε απώλεια πληροφορίας στα άκρα.

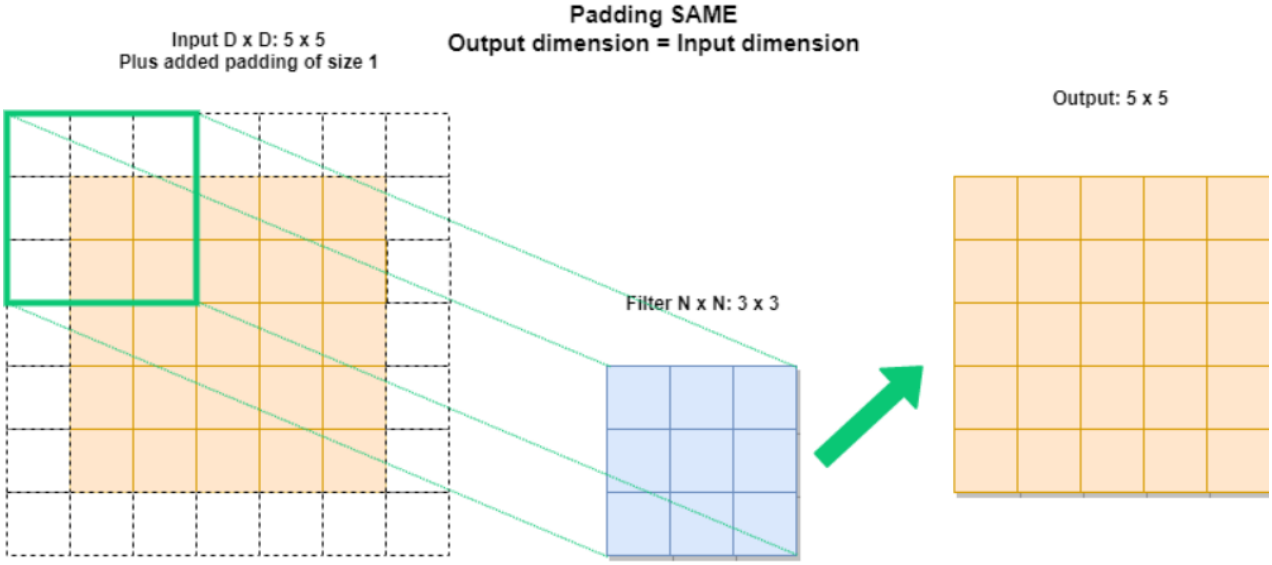
Για να αντιμετωπιστεί αυτό το πρόβλημα, χρησιμοποιείται το padding, δηλαδή προστίθενται επιπλέον pixels γύρω από την εικόνα, πριν εφαρμοστεί η συνέλιξη.

Οι δύο κύριες μορφές padding είναι:

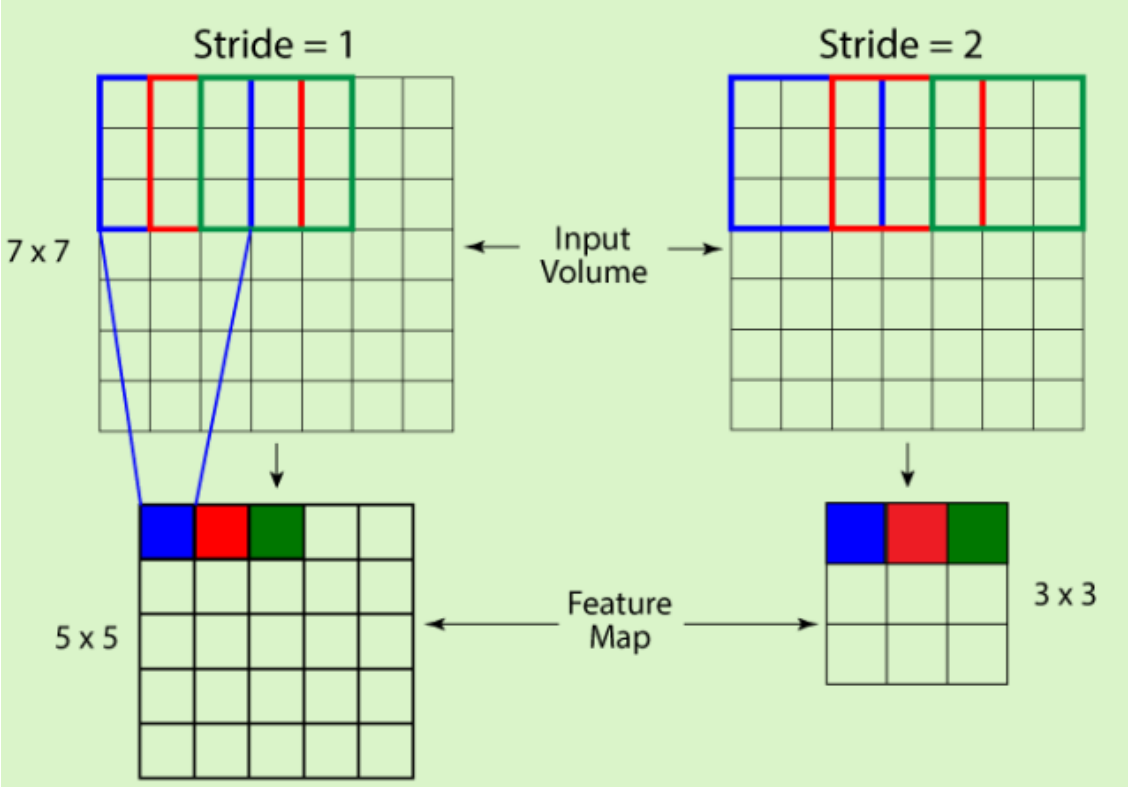
Valid: Δεν προστίθενται επιπλέον pixels, οπότε το φίλτρο εφαρμόζεται ακριβώς πάνω στην είσοδο. Αυτό οδηγεί σε μείωση των διαστάσεων του χάρτη χαρακτηριστικών.

Same: Προστίθενται αρκετά επιπλέον pixels έτσι ώστε η έξοδος να έχει τις ίδιες διαστάσεις με την είσοδο.

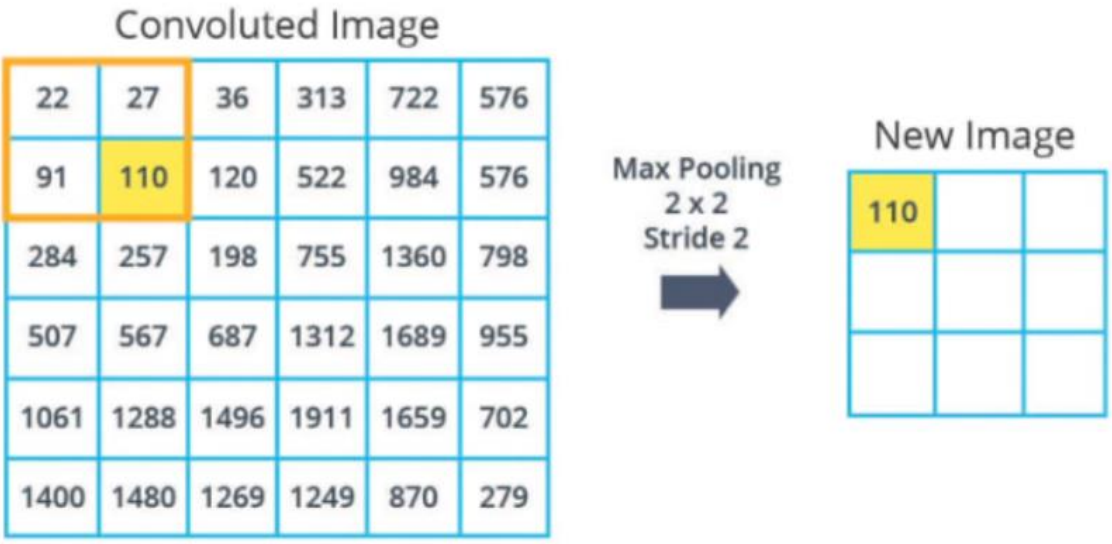


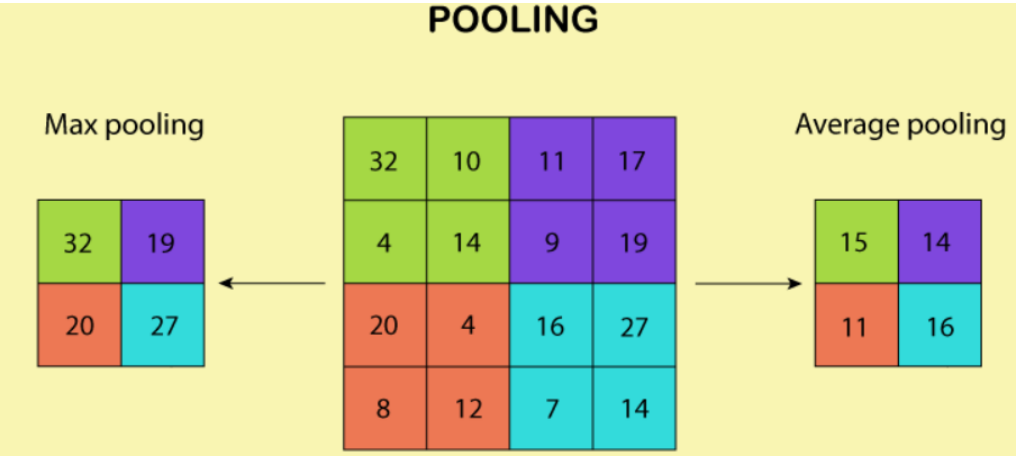


Το stride (βήμα) καθορίζει πόσα pixels κινείται το φίλτρο κάθε φορά που πραγματοποιείται η μετακίνηση του. Για παράδειγμα, με ένα stride ίσο με 1, το φίλτρο κινείται κατά ένα pixel κάθε φορά. Αν το stride είναι 2, το φίλτρο θα μετακινείται κατά δύο pixels και κάθετα και οριζόντια. Η χρήση μεγαλύτερου stride μειώνει τις διαστάσεις του χάρτη χαρακτηριστικών.



Η λειτουργία του pooling αναφέρεται στη μείωση των διαστάσεων ενός χάρτη χαρακτηριστικών με το να επιλέγει τη μέγιστη τιμή ή τον μέσο όρο των τιμών σε κάποιες περιοχές. Τα δημοφιλέστερα είδη pooling είναι το Max Pooling και το Average Pooling.





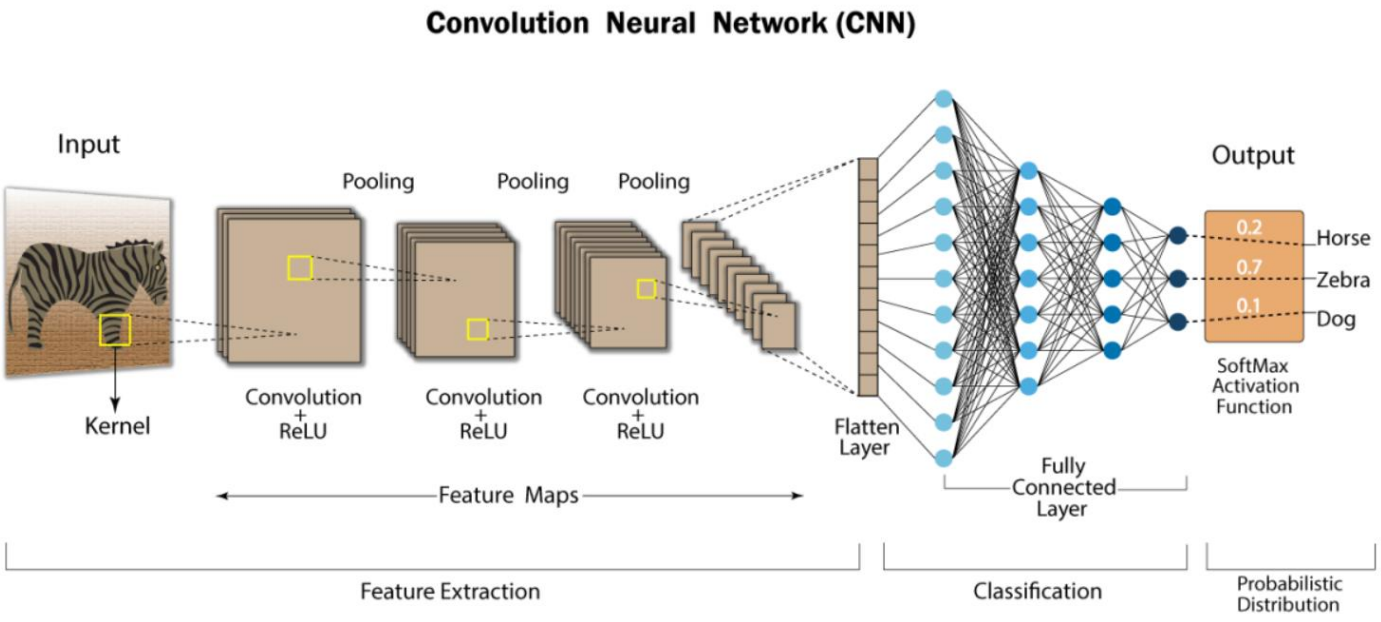
Τα **συνελικτικά νευρωνικά δίκτυα (Convolutional Neural Networks ή CNNs)** είναι ένα είδος τεχνητών νευρωνικών δικτύων που σχεδιάστηκε ειδικά για την επεξεργασία εικόνων και παρόμοιων δομημένων δεδομένων. Τα CNNs έχουν αποδειχθεί εξαιρετικά αποτελεσματικά στον τομέα της όρασης υπολογιστών.

Βασικά χαρακτηριστικά των CNNs:

Συνελικτικά Επίπεδα: Χρησιμοποιούν συνελικτικά επίπεδα για την εξόρυξη χαρακτηριστικών από τις εικόνες. Αυτά τα επίπεδα εκτελούν συνελίξεις για ανίχνευση χαρακτηριστικών.

Επίπεδα Μείωσης των διαστάσεων: Συνήθως περιλαμβάνουν pooling επίπεδα, όπως το Max Pooling, για τη μείωση της χωρικής διάστασης των χαρακτηριστικών και τον έλεγχο του overfitting.

Πλήρως Διασυνδεδεμένα Επίπεδα: Συνήθως, τα τελευταία επίπεδα του δικτύου είναι πλήρως διασυνδεδεμένα, συνδέοντας τα χαρακτηριστικά που εξήχθησαν με τις επιθυμητές εξόδους.



Η συνάρτηση ενεργοποίησης ReLU (Rectified Linear Unit) είναι μια μη γραμμική συνάρτηση που ευρέως χρησιμοποιείται σε συνελικτικά νευρωνικά δίκτυα (CNNs) και άλλα είδη τεχνητών νευρωνικών δικτύων. Επιστρέφει την ίδια τιμή, αν η είσοδος είναι θετική και το μηδέν αν η είσοδος είναι αρνητική. Μαθηματικά, αυτό μπορεί να εκφραστεί ως f(x)=max(0,x). Επειδή είναι μια μη γραμμική συνάρτηση, επιτρέπει στα νευρωνικά δίκτυα να μάθουν πιο πολύπλοκα χαρακτηριστικά και προτιμάται σε σχέση με γραμμικές συναρτήσεις ενεργοποίησης.

Στα συνελικτικά νευρωνικά δίκτυα, η διαδικασία της εκπαίδευσης περιλαμβάνει την προσαρμογή των παραμέτρων, όπως τα φίλτρα και τα bias, ώστε το δίκτυο να μπορεί να εκτελεί καλύτερα τις επιθυμητές εργασίες.

Όταν το μοντέλο εκπαιδευτεί επαρκώς, τα φίλτρα και το bias μπορούν να ανιχνεύουν χαρακτηριστικά σε δεδομένα που δεν είχαν δει κατά τη διάρκεια της εκπαίδευσης, και το μοντέλο γίνεται ικανό να γενικεύει αποτελεσματικά σε νέα δεδομένα.

Παρουσίαση σειριακού κώδικα

Για την εκτέλεση του convolution και max pooling, χρησιμοποιήθηκαν τρείς συναρτήσεις.

Αρχικά, η συνάρτηση convolution εφαρμόζει συνέλιξη σε έναν πίνακα input διάστασης hxw, με τη χρήση ενός φίλτρου μεγέθους kernel_size x kernel_size. Το τελικό feature map αποθηκεύεται στον πίνακα output. Το άθροισμα των πολλαπλασιασμών του kernel με την αντίστοιχη θέση του αρχικού πίνακα, είναι το νέο στοιχείο του τελικού feature map. Τα όρια στις for έχουν οριστεί έτσι ώστε να έχουμε valid padding, δηλαδή το kernel δε βγαίνει εκτός των ορίων της εικόνας. Η τελική εικόνα έχει διάσταση μικρότερη από την αρχική (για kernel 3x3 μικρότερη κατά 2 στο ύψος και στο πλάτος). Το βήμα στην μετακίνηση του φίλτρου (stride) είναι ένα.

```
void convolution(int **input, double **output, int h, int w, int kernel_size, double kernel[][3])
{
    int i, j, x, y;
    double sum;
    int kernelCenter = kernel_size / 2.0;

    for (i = kernelCenter; i < h - kernelCenter; i++)
    {
        for (j = kernelCenter; j < w - kernelCenter; j++)
        {
            sum = 0;
            for (x = 0; x < kernel_size; x++)
            {
                for (y = 0; y < kernel_size; y++)
                {
                    sum += kernel[x][y] * (double)(input[i - kernelCenter + x][j - kernelCenter + y]);
                }
            }
            output[i][j] = sum;
        }
    }
}
```

Η συνάρτηση convolution2D παίρνει σαν είσοδο τρεις πίνακες (τα τρία κανάλια RGB της εικόνας) και τρία kernel (ένα για το κάθε κανάλι), παράγει τρία διαφορετικά feature map εφαρμόζοντας συνέλιξη ανεξάρτητα στα τρία κανάλια και προσθέτει τα τρία feature map για την παραγωγή του τελικού feature map. Επίσης, στον τελικό πίνακα προστίθεται το bias και εφαρμόζεται συνάρτηση ενεργοποίησης ReLU.

```
void convolution2D(int **input1, int **input2, int **input3, int h, int w, int kernel_size, double kernel1[][3], double kernel2[][3],
                   double kernel3[][3], double bias, double **output1, double **output2, double **output3, int **output_final)
{
    int i, j;
    double res;

    // Perform convolutions
    convolution(input1, output1, h, w, kernel_size, kernel1);
    convolution(input2, output2, h, w, kernel_size, kernel2);
    convolution(input3, output3, h, w, kernel_size, kernel3);

    for (i = 0; i < h; i++)
    {
        for (j = 0; j < w; j++)
        {
            res = output1[i][j] + output2[i][j] + output3[i][j] + bias;

            // ReLU Activation Function
            if(res>0)
            {
                output_final[i][j] = round(res);
            }
            else
            {
                output_final[i][j] = 0;
            }
        }
    }
}
```

Η συνάρτηση maxpooling παίρνει σαν είσοδο input τον πίνακα που προέκυψε από τη συνέλιξη των τριών καναλιών της εικόνας. Στη συνέχεια, εφαρμόζεται maxpooling και οι τιμές αποθηκεύονται στον πίνακα output, ο οποίος αποτελεί και το τελικό αποτέλεσμα του αλγορίθμου. Εμείς ορίσαμε kernel_size=2 και stride=2, αυτό σημαίνει ότι η διάσταση της τελικής εικόνας μειώνεται στο μισό σε σχέση με την αρχική. Επίσης, κάθε φορά ελέγχουμε το παράθυρο να μην βγαίνει εκτός της εκόνας.

```
void maxpooling(int **input, int **output, int h, int w, int kernel_size, int stride)
{
    int i, j, x, y;
    int kernelCenter = kernel_size / 2.0;

    for (i = 0; i < h; i += stride)
    {
        for (j = 0; j < w; j += stride)
        {
            int max = input[i][j];

            for (x = 0; x < kernel_size; x++)
            {
                for (y = 0; y < kernel_size; y++)
                {
                    int curr_i = i - kernelCenter + x;
                    int curr_j = j - kernelCenter + y;

                    if (curr_i >= 0 && curr_i < h && curr_j >= 0 && curr_j < w)
                    {
                        int current = input[curr_i][curr_j];
                        if (current > max)
                        {
                            max = current;
                        }
                    }
                }
            }

            output[i / stride][j / stride] = max;
        }
    }
}
```

Παρακάτω φαίνεται ένα τυχαίο φίλτρο που χρησιμοποιήσαμε και το αντίστοιχο bias.

```
// Kernels
kernel_size = 3;
pooling_kernel_size = 2;
stride = 2;

double kernel1[3][3] =
{
    {-0.23682003, 0.1700454, 0.09052496},
    {-0.00821521, 0.1779402, -0.09653653},
    {0.07029217, -0.12574358, 0.10022978}
};

double kernel2[3][3] =
{
    {-0.07746775, -0.09846717, 0.03488337},
    {0.00120761, 0.07884538, -0.07599071},
    {0.05539479, -0.03348019, -0.07456464}
};

double kernel3[3][3] =
{
    {0.01844125, 0.20088513, -0.04941435},
    {0.13128215, -0.09104527, 0.06280853},
    {-0.05294552, 0.10650568, -0.09848029}
};

double bias = -3.9732606;

int kernelCenter = kernel_size / 2.0;
```

Στον σειριακό κώδικα απλά καλούνται οι συναρτήσεις και γίνονται μετρήσεις για το χρόνο.

```
////// Serial code
double start_time, end_time, elapsed_time_s;
start_time = omp_get_wtime();
convolution2D((int **)red_channel, (int **)green_channel, (int **)blue_channel, image_height, image_width, kernel_size,
              kernel1, kernel2, kernel3, bias, outputR, outputG, outputB, result_serial);
maxpooling(result_serial, result_serial_final, image_height, image_width, pooling_kernel_size, stride);

end_time = omp_get_wtime();

elapsed_time_s = end_time - start_time;
printf("\nElapsed time Serial: %fs\n\n", elapsed_time_s);
```

Μεθοδολογία της παράλληλης επίλυσης

Στη συνάρτηση convolution_parallel για την παραλληλοποίηση χρησιμοποιήθηκε η εντολή collapse. Οι δύο εξωτερικές for μπορούν να εκτελεστούν παράλληλα, γιατί για να παραχθεί το κάθε νέο στοιχείο του feature map χρησιμοποιούμε τιμές από τον αρχικό πίνακα. Οπότε, η αποθήκευση δεν μπορεί να γίνει στον αρχικό πίνακα, επειδή δε θέλουμε να τον αλλάξουμε. Πέρα από αυτό όλα τα στοιχεία μπορούν να υπολογιστούν παράλληλα (δε υπάρχουν εξαρτήσεις και συνθήκες ανταγωνισμού).

Οι εσωτερικές for θεωρήθηκε μη αποδοτικό να παραλληλοποιηθούν, επειδή συνήθως τα kernel έχουν μικρό μέγεθος (πχ, 3x3=9 στοιχεία), οπότε το επιπλέον overhead για την παραλληλοποίηση θα υπερβεί το χρόνο που κερδίζουμε. Με reduction στο sum θα μπορούσε να παραλληλοποιηθεί.

```
void convolution_parallel(int **input, double **output, int h, int w, int kernel_size, double kernel[][3], int P)
{
    int i, j, x, y;
    int kernelCenter = kernel_size / 2.0;

    # pragma omp parallel for private(i, j, x, y) num_threads(P) collapse(2)
    for (i = kernelCenter; i < h - kernelCenter; i++)
    {
        for (j = kernelCenter; j < w - kernelCenter; j++)
        {
            double sum = 0; // sum is private
            for (x = 0; x < kernel_size; x++)
            {
                for (y = 0; y < kernel_size; y++)
                {
                    sum += kernel[x][y] * (double)(input[i - kernelCenter + x][j - kernelCenter + y]);
                }
                output[i][j] = sum;
            }
        }
    }
}
```

Στη συνάρτηση convolution2D_parallel για την παραλληλοποίηση επίσης χρησιμοποιήθηκε η εντολή collapse. Οι δύο εξωτερικές for μπορούν να εκτελεστούν παράλληλα, καθώς για κάθε νέο pixel αθροίζουμε τα στοιχεία των τριών πινάκων, προσθέτουμε μια σταθερά και κάνουμε και έναν έλεγχο (σε όλα αυτά τα βήματα δεν υπάρχουν εξαρτήσεις).

Επίσης, δοκιμάσαμε να παραλληλοποιήσουμε και την κλήση της συνάρτησης convolution_parallel που γίνεται τρεις φορές, όμως πήραμε χειρότερο αποτέλεσμα, πιθανώς επειδή αυτή η συνάρτηση εκτελείται ήδη παράλληλα. Πάντως το αποτέλεσμα ήταν σωστό, γιατί το feature map κάθε καναλιού αποθηκεύεται σε έναν νέο πίνακα.

```
void convolution2D_parallel(int **input1, int **input2, int **input3, int h, int w, int kernel_size, double kernel1[][3], double kernel2[][3],
                            double kernel3[][3], double bias, double **output1, double **output2, double **output3, int **output_final, int P)
{
    int i, j;
    double res;

    // Perform convolutions
    convolution_parallel(input1, output1, h, w, kernel_size, kernel1, P);
    convolution_parallel(input2, output2, h, w, kernel_size, kernel2, P);
    convolution_parallel(input3, output3, h, w, kernel_size, kernel3, P);

    # pragma omp parallel private(i, j, res) num_threads(P)
    {
        # pragma omp for collapse(2)
        for (i = 0; i < h; i++)
        {
            for (j = 0; j < w; j++)
            {
                res = output1[i][j] + output2[i][j] + output3[i][j] + bias;

                // ReLU Activation Function
                if(res>0)
                {
                    output_final[i][j] = round(res);
                }
                else
                {
                    output_final[i][j] = 0;
                }
            }
        }
    }
}
```

Στη συνάρτηση maxpooling_parallel για την παραλληλοποίηση επίσης χρησιμοποιήθηκε η εντολή collapse. Οι δύο εξωτερικές for μπορούν να εκτελεστούν παράλληλα. Ισχύει ότι ακριβώς αναφέραμε και για την παραλληλοποίηση της convolution_parallel.

```
void maxpooling_parallel(int **input, int **output, int h, int w, int kernel_size, int stride, int P)
{
    int i, j, x, y;
    int kernelCenter = kernel_size / 2.0;

    # pragma omp parallel for private(i, j, x, y) num_threads(P) collapse(2)
    for (i = 0; i < h; i += stride)
    {
        for (j = 0; j < w; j += stride)
        {
            int max = input[i][j];

            for (x = 0; x < kernel_size; x++)
            {
                for (y = 0; y < kernel_size; y++)
                {
                    int curr_i = i - kernelCenter + x;
                    int curr_j = j - kernelCenter + y;

                    if (curr_i >= 0 && curr_i < h && curr_j >= 0 && curr_j < w)
                    {
                        int current = input[curr_i][curr_j];
                        if (current > max)
                        {
                            max = current;
                        }
                    }
                }
            }

            output[i / stride][j / stride] = max;
        }
    }
}
```

Στον παράλληλο κώδικα απλά καλούνται οι συναρτήσεις και γίνονται μετρήσεις για το χρόνο. Επίσης, υπολογίζεται το speedup και το efficiency.

```
////// Parallel code
int P;
double speedup, elapsed_time, efficiency;
for (P = 1; P <= 64; P = P*2)
{
    // Restore the initial values in the matrices
    for(i=0; i<image_height; i++)
    {
        for(j=0; j<image_width; j++)
        {
            result_parallel[i][j] = 0;
            result_parallel_final[i][j] = 0;
            outputR[i][j] = 0;
            outputG[i][j] = 0;
            outputB[i][j] = 0;
        }
    }

    printf("P = %d\n", P);
    start_time = omp_get_wtime();
    convolution2D_parallel((int **)red_channel, (int **)green_channel, (int **)blue_channel, image_height, image_width, kernel_size,
                           kernel1, kernel2, kernel3, bias, outputR, outputG, outputB, result_parallel, P);
    maxpooling_parallel(result_parallel, result_parallel_final, image_height, image_width, pooling_kernel_size, stride, P);

    end_time = omp_get_wtime();

    elapsed_time = end_time - start_time;
    printf("Elapsed time Parallel: %fs\n", elapsed_time);
    speedup = elapsed_time_s/elapsed_time;
    printf("Speedup: %f\n", speedup);
    efficiency = speedup/P;
    printf("Efficiency: %f\n", efficiency);
```

Περιγραφή των πειραματικών αποτελεσμάτων

Αρχικά, να αναφέρουμε πως η περιγραφή των πειραματικών αποτελεσμάτων έγινε για δυο περιπτώσεις. Αυτές είναι η παρακάτω:

1. Εφαρμογή του φίλτρου σε πραγματικές εικόνες RGB διάφορων μεγεθών

2. Εφαρμογή του φίλτρου σε τυχαίους πίνακες των χρωματικών καναλιών RGB επιλέγοντας μεγαλύτερα μεγέθη προβλήματος για θεωρητική ανάλυση

1η περίπτωση:

Ο αλγόριθμος εφαρμόστηκε σε διάφορα μεγέθη εικόνων, και πιο συγκεκριμένα στα παρακάτω 4 μεγέθη προβλήματος.

1. Elephant (2040 x 1356)px

2. Flower (4000 x 3000)px

3. Landscape (8000 x 6000)px

4. Rainbow (10000 x 7357)px

Το πρώτο βήμα είναι να κατεβάσουμε τις εικόνες από το διαδίκτυο στις διαστάσεις που αναγράφονται και έπειτα να τις μετατρέψουμε από ‘ .jpg ’ σε ‘ .raw ‘ αρχεία για να μπορούμε να τις διαβάσουμε σαν πίνακες των καναλιών RGB. Έτσι λοιπόν, ως input θα έχουμε τρείς δισδιάστατους πίνακες, matrixR, matrixG και matrixB, με διαστάσεις width x height της αντίστοιχης εικόνας.

Αφού τρέξαμε τον αλγόριθμο που περιεγράφηκε παραπάνω για κάθε εικόνα και για κάθε πολλαπλάσιο του δύο των διαθέσιμων πυρήνων του μηχανήματος pxeon2, πήραμε σαν output το ‘ .raw ‘ αρχείο της τελικής μορφοποιημένης εικόνας αλλά και το ‘ .raw ‘ αρχείο της εικόνας πριν την εφαρμογή του max pooling (δηλαδή το feature map).

Τελευταίο βήμα αποτελεί η μεταμόρφωση των ‘ .raw ‘ αρχείων σε ‘ .png ‘ αρχεία για να μπορέσουμε να διακρίνουμε τα οπτικά αποτελέσματα. Παρακάτω φαίνονται τα αποτελέσματα των εικόνων 2 (flower) και 3 (landscape).

Feature_map_flower.png

Feature_map_after_pooling_flower.png

Feature_map_landscape.png

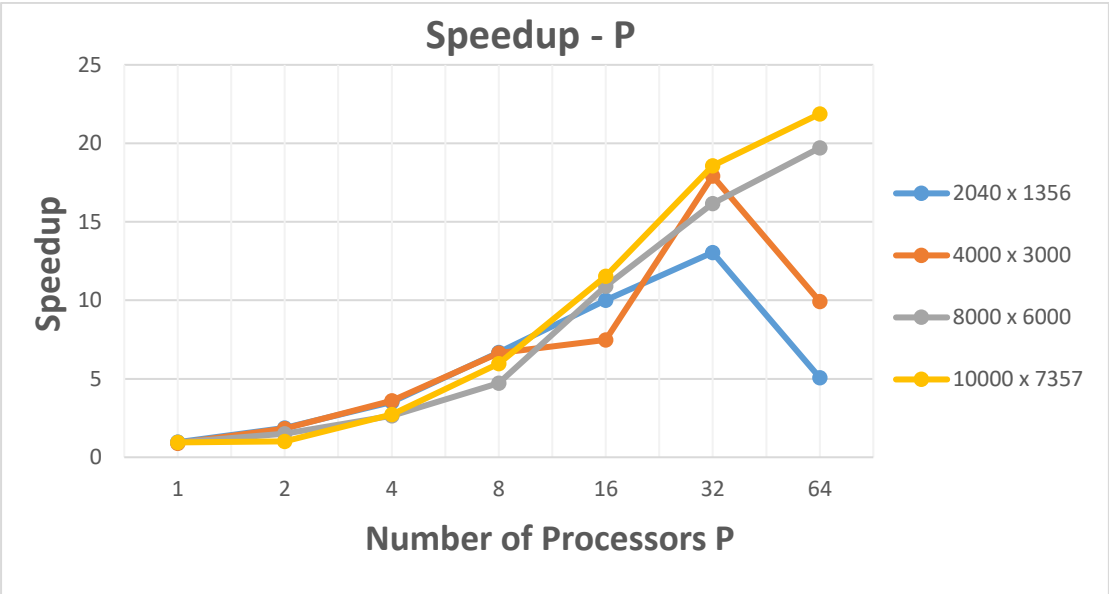
Feature_map_after_pooling_landscape.png

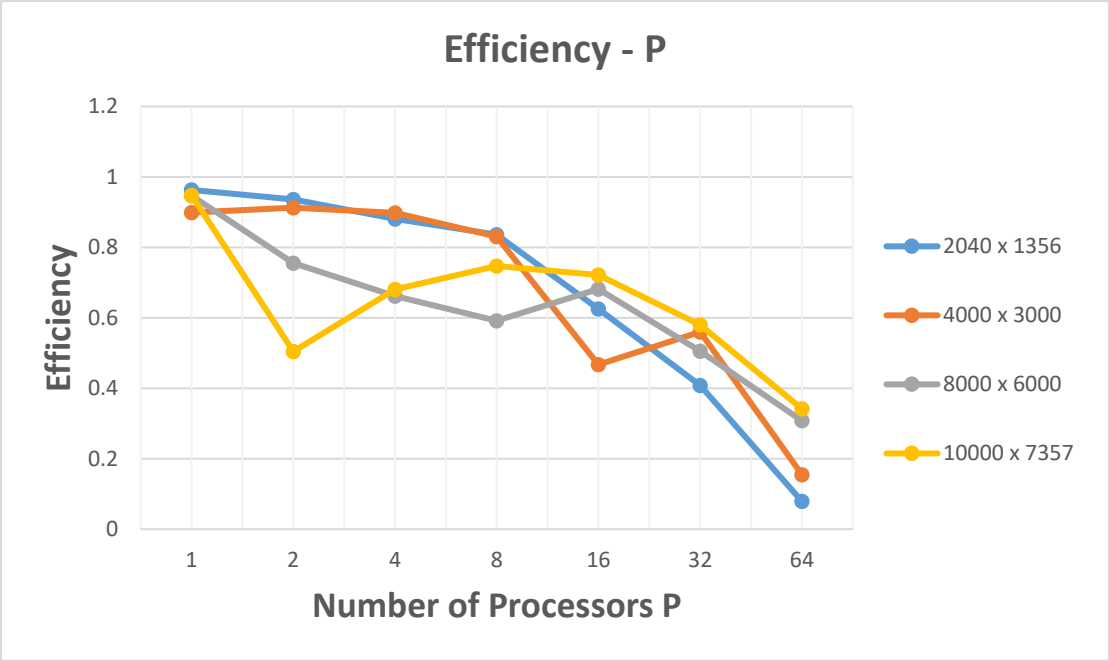
Όπως μπορούμε να παρατηρήσουμε, το output μας δίνεται σε grayscale χρωματισμό και ιδιαίτερα με την εφαρμογή του max pooling, στην τελική εικόνα επιλέγονται να τονιστούν μονό οι μεγαλύτερες τιμές του αντίστοιχου 2x2 kernel, τονίζοντας τα έντονα pixel περισσότερο.

Πέρα από τα οπτικά αποτελέσματα, μετρήσαμε τον χρόνο εκτέλεσης του σειριακού υπολογισμού του αποτελέσματος καθώς και της παράλληλης υλοποίησης που αναφέρθηκε λεπτομερώς νωρίτερα. Επίσης υπολογίσαμε την επιτάχυνση (speedup) και την αποδοτικότητα (efficiency) σε κάθε βήμα και τα αποτελέσματα φαίνονται παρακάτω. Με κίτρινο υπογραμμίζονται οι υψηλότερες τιμές του speedup που πετύχαμε.

Dimensions (WxH)	Metric	Serial	1	2	4	8	16	32	64
2040 x 1356	Time	0,30468	0,316337	0,162725	0,086467	0,045545	0,030449	0,023349	0,060083
	Speedup		0,96315006	1,872361	3,523656	6,689648	10,00624	13,04895	5,070985
	Efficiency		0,96315006	0,936181	0,880914	0,836206	0,62539	0,40778	0,079234
4000 x 3000	Time	1,267015	1,408854	0,694056	0,352545	0,190607	0,169574	0,070728	0,12769
	Speedup		0,89932314	1,825523	3,59391	6,647264	7,471753	17,91391	9,922586
	Efficiency		0,89932314	0,912761	0,898477	0,830908	0,466985	0,55981	0,15504
8000 x 6000	Time	5,197924	5,48847	3,439201	1,96453	1,098368	0,476525	0,32152	0,263694
	Speedup		0,94706248	1,511375	2,645887	4,732407	10,90798	16,16672	19,71195
	Efficiency		0,94706248	0,755688	0,661472	0,591551	0,681749	0,50521	0,307999
10000 x 7357	Time	8,082142	8,524719	8,013069	2,967973	1,351805	0,700017	0,435144	0,369425
	Speedup		0,9480831	1,00862	2,723118	5,978778	11,54564	18,57349	21,87763
	Efficiency		0,9480831	0,50431	0,68078	0,747347	0,721602	0,580422	0,341838

Παρουσιάζονται επίσης και τα διαγράμματα επιτάχυνσης και αποδοτικότητας για άμεση σύγκριση της παραλληλίας με γνώμονα το πλήθος των πυρήνων.





2η περίπτωση:

Σε αυτή την περίπτωση, όπως προαναφέραμε, δημιουργούμε εμείς τυχαία πίνακες με τις διαστάσεις που θέλουμε με σκοπό να επαληθεύσουμε τα αποτελέσματα των πράξεων αλλά και να παρατηρήσουμε τα αποτελέσματα σε μεγαλύτερα μεγέθη προβλήματος.

Ο προφανής λόγος για αυτό είναι η έλλειψη δυνατοτήτων των δικών μας υπολογιστών για την γρήγορη και εύκολη μεταχείριση των εικόνων μεγάλου μεγέθους. Για αυτό λοιπόν, ο αλγόριθμος αυτός είναι ίδιος με τον αρχικό αλλά αλλάζει το width και το height που επιλέγουμε κάθε φόρα ως διαστάσεις για τους τυχαία ορισμένους πίνακες.

Αυτό μας επιτρέπει να εισάγουμε μεγέθη προβλήματος όπως τα παρακάτω:

1. Width x Height = **8000 x 8000** pixel
2. Width x Height = **16000 x 16000** pixel
3. Width x Height = **32000 x 32000** pixel
4. Width x Height = **50000 x 50000** pixel

Πριν περάσουμε στην εκτέλεση των παραπάνω όμως, θα ορίσουμε μικρές διαστάσεις Width = 4 και Height = 6, για να μπορούμε να επαληθεύσουμε το σωστό αποτέλεσμα των πράξεων μεταξύ σειριακού και παράλληλου κώδικα. Παρακάτω φαίνονται τα αποτελέσματα των πινάκων.

```
Featuremap Serial:
56      26      0       59
19      0       34      30
21      0       113     0
0       40      39      5
92      0       33      33
0       72      50      72

Featuremap Parallel:
56      26      0       59
19      0       34      30
21      0       113     0
0       40      39      5
92      0       33      33
0       72      50      72

Featuremap Serial After Pooling:
56      59
40      113
92      72

Featuremap Parallel After Pooling:
56      59
40      113
92      72
```

Βλέπουμε ότι τα αποτελέσματα σειριακού και παράλληλου κώδικα είναι ίδια. Επιπλέον, με σκοπό να επαληθεύσουμε ότι οι τιμές που βγάζουν οι αλγόριθμοι από τις πράξεις του φίλτρου και την εφαρμογή του max pooling, συνθέσαμε τον ίδιο αλγόριθμο σε python χρησιμοποιώντας τις ήδη έτοιμες συναρτήσεις convolve2D και correlate2D. Τα αποτελέσματα που πήραμε για τις ίδιες εισόδους καναλιών RGB, όπως και για τα ίδια kernel και bias ήταν ίδια.

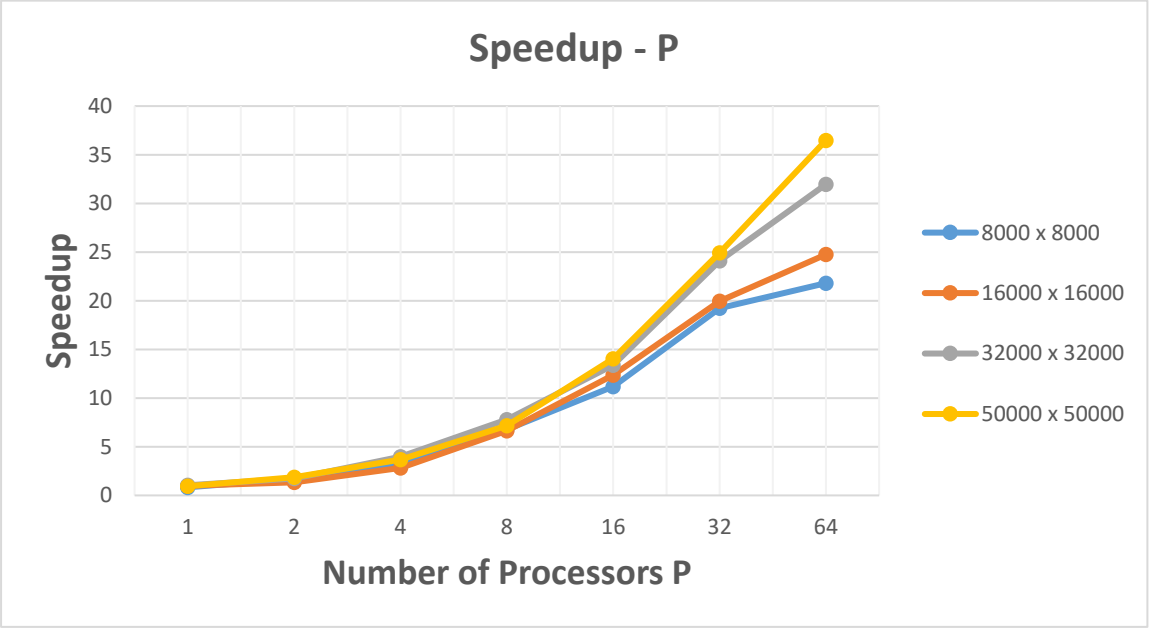
```python
# Perform 2D convolutions with bias
red_result = correlate2d(matrixR, kernel1, mode='valid')
green_result = correlate2d(matrixG, kernel2, mode='valid')
blue_result = correlate2d(matrixB, kernel3, mode='valid')

# Combine the results by element-wise addition
# feature_map = red_result + green_result + blue_result + bias
feature_map = np.round(red_result + green_result + blue_result + bias).astype(int)
feature_map[feature_map < 0] = 0
print(feature_map)
```

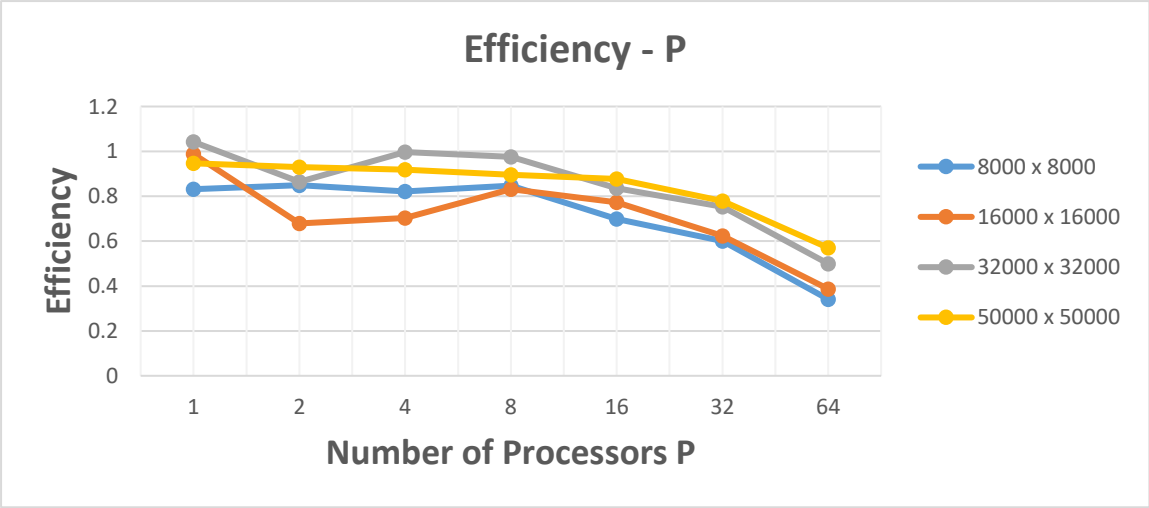
```
[[ 56  26   0  59]
 [ 19   0  34  30]
 [ 21   0 113   0]
 [  0  40  39   5]
 [ 92   0  33  33]
 [  0  72  50  72]]
```

Συνεπώς τα αποτελέσματα των πράξεων μας είναι σωστά.

Αυξάνοντας το μέγεθος του προβλήματος στις τιμές που αναφέρθηκαν παραπάνω παίρνουμε ένα νέο πίνακα με μετρήσεις χρόνου, επιτάχυνσης και αποδοτικότητας καθώς και τα αντίστοιχα διαγράμματα.

Dimensions (WxH)	Metric	Serial	1	2	4	8	16	32	64
8000 x 8000	Time	7,691255	9,252673	4,527534	2,340335	1,134101	0,688172	0,400006	0,352674
	Speedup		0,83124682	1,698774	3,286391	6,781808	11,17636	19,22785	21,8084
	Efficiency		0,83124682	0,849387	0,821598	0,847726	0,698522	0,60087	0,340756
16000 x 16000	Time	30,455921	30,824143	22,44841	10,81734	4,581188	2,464383	1,52622	1,230298
	Speedup		0,9880541	1,356707	2,815474	6,64804	12,35844	19,95513	24,75491
	Efficiency		0,9880541	0,678354	0,703868	0,831005	0,772402	0,623598	0,386796
32000 x 32000	Time	139,36463	133,718669	80,59584	34,9455	17,86592	10,41564	5,782209	4,360229
	Speedup		1,04222271	1,729179	3,988057	7,800587	13,38032	24,10232	31,96269
	Efficiency		1,04222271	0,864589	0,997014	0,975073	0,83627	0,753197	0,499417
50000 x 50000	Time	317,79399	335,651445	170,9095	86,50161	44,36099	22,64507	12,74653	8,709758
	Speedup		0,94679763	1,859428	3,673851	7,163817	14,0337	24,93181	36,48712
	Efficiency		0,94679763	0,929714	0,918463	0,895477	0,877106	0,779119	0,570111





Συμπεράσματα και πολυπλοκότητα αλγορίθμου

Αρχικά, παρατηρούμε πως όσο αυξάνεται το μέγεθος του προβλήματος τόσο αυξάνεται και το speedup. Έτσι συμπεραίνουμε ότι εάν είχαμε μεγαλύτερες εικόνες και μεγαλύτερο πλήθος επεξεργαστών, η παραλληλοποίηση θα γινόταν όλο και πιο γρήγορη σε σχέση με την σειριακή υλοποίηση. Βέβαια, είναι συχνό για έναν αλγόριθμο που τρέχει σε 32 πυρήνες για παράδειγμα να είναι πιο γρήγορος από ότι να εκτελείται σε 64 πυρήνες για ένα συγκεκριμένο μέγεθος προβλήματος. Παρόλα αυτά, με τα παραπάνω αποτελέσματα διαπιστώνουμε ότι το πρόβλημα μας είναι πλήρως παραλληλοποιήσιμο.

Επίσης παρατηρούμε ότι καθώς αυξάνουμε το πλήθος των νημάτων και το μέγεθος προβλήματος, η απόδοση παραμένει σχεδόν σταθερή. Συνεπώς, το πρόβλημα αυτό είναι επεκτάσιμο.

Όσον αφορά την πολυπλοκότητα η συνέλιξη απαιτεί O(H×W×C×K×K), όπου

Μέγεθος εικόνας εισόδου (H x W x C)

Kernel Size (K x K)

Άρα σε εμάς για K=3 και C=3 κανάλια η πολυπλοκότητα είναι O(H×W×27) και αν H~W~N, όπου N θεωρούμε ότι είναι το μέγεθος του προβλήματος η πολυπλοκότητα είναι O(N^2).

Όσον αφορά την πολυπλοκότητα το Pooling απαιτεί O(H/P×W/P×C×P×P), όπου

Μέγεθος εικόνας εισόδου (H x W x C)

Pooling Kernel Size (P × P)

Άρα σε εμάς για K=2 και C=3 κανάλια η πολυπλοκότητα είναι O(H×W×3) και αν H~W~N, όπου N θεωρούμε ότι είναι το μέγεθος του προβλήματος η πολυπλοκότητα είναι O(N^2).

Οπότε, η συνολική πολυπλοκότητα είναι **O(N^2)**.

Η πολυπλοκότητα του παράλληλου κώδικα εξαρτάται από τον αριθμό των νημάτων P που χρησιμοποιούνται και το μέγεθος της εικόνας εισόδου N. Υποθέτοντας ότι το N είναι αρκετά μεγάλο, η χρονική πολυπλοκότητα του παράλληλου κώδικα μπορεί να προσεγγιστεί ως **O(N^2/P)**, που αντιπροσωπεύει τον μειωμένο χρόνο υπολογισμού λόγω της παραλληλοποίησης.

*Στο αρχείο Project1.c βρίσκεται ο κώδικας στον οποίο ελέγχουμε με τυχαίους πίνακες αν τα αποτελέσματά μας είναι σωστά. Στο Project1_final.c βρίσκεται ο κώδικας που διαχειρίζεται τις εικόνες. Το Project1_v2.c βρίσκεται ο κώδικας για τυχαίους πίνακες αλλά για μεγαλύτερα μεγέθη προβλήματος.

Βιβλιογραφία

https://medium.com/analytics-vidhya/convolution-padding-stride-and-pooling-in-cnn-13dc1f3ada26

https://poloclub.github.io/cnn-explainer/