

# Όραση Υπολογιστών

2020-2021



Εργασία 4η

**Καρυπίδης Ευστάθιος 57556**

Στην παρακάτω τεχνική αναφορά περιγράφεται η λύση της 4ης εργασίας, όπου σε συνέχεια της 3ης εργασίας έχει ως στόχο την ταξινόμηση πολλών κλάσεων(multi-class classification), αλλά αυτήν την φορά με χρήση νευρωνικών δικτύων. Ειδικότερα, ζητείται να υλοποιηθούν δύο αρχιτεκτονικές συνελικτικλων δικτύων σε python με χρήση της βιβλιοθήκης Keras-Tensorflow, μια με χρήση μή προ-εκπαίδευμένου δικτύου και μία με χρήση transfer learning δηλαδή ενός προεκπαίδευμένου δικτύου. Αφού γίνει αναφορά στο απαραίτητο θεωρητικό υπόβαθρο, θα αιτιολογηθεί η χρήση των διαφόρων τιμών σε παραμέτρους και τεχνικές που χρησιμοποιήθηκαν καθώς και θα παρουσιαστούν τα τελικά αποτελέσματα διαφόρων δοκιμών.

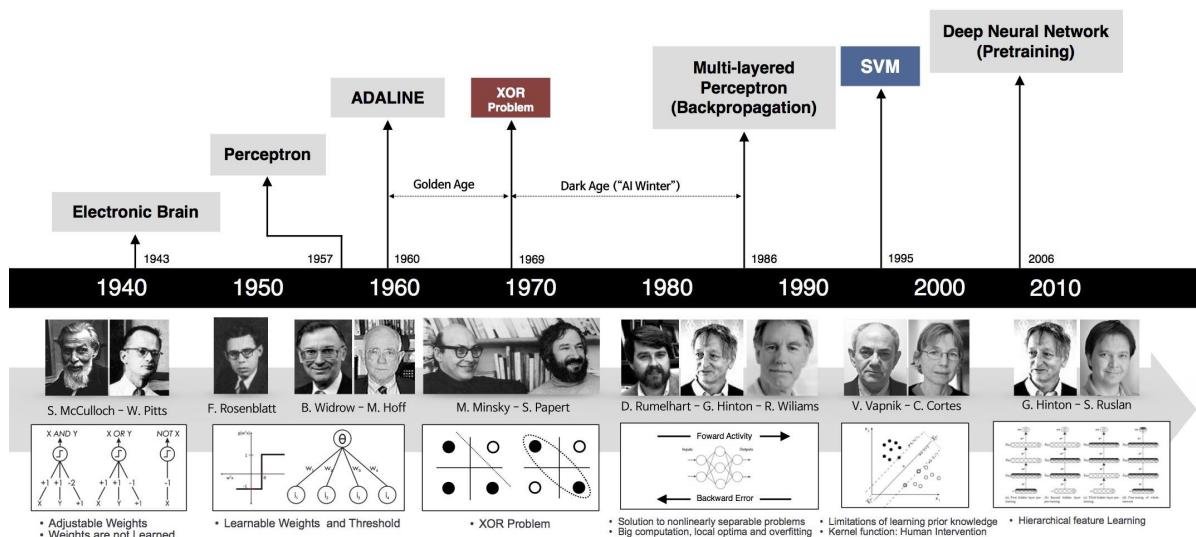
# Περιγραφή προβλήματος - Θεωρητική ανάλυση

Για την υλοποίηση της εργασίας δόθηκαν δύο βάσεις εικόνων "imagedb" και "imagedb\_test" οι οποίες αποτελούν υποσύνολο της βάσης εικόνων <https://btst.ethz.ch/shareddata/> (BelgiumTS Dataset), όπου η πρώτη θα χρησιμοποιηθεί για την εκπαίδευση(training set) του συστήματος και η δεύτερη για την δοκιμή και αξιολόγηση του(test set). Ωστόσο, για την εκπαίδευση των δικτύων είναι απαραίτητο και ένα validation set. Για το σκοπό αυτό θα περιγραφεί παρακάτω η διαδικασία για παραγωγή του συγκεκριμένου συνόλου.

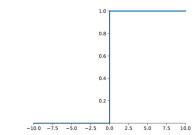
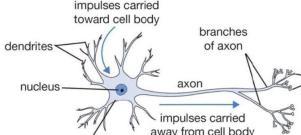
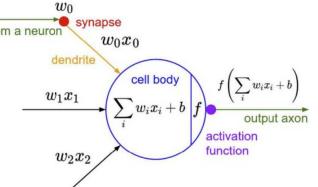
Στην προηγούμενη εργασία δοκιμάστηκαν στο συγκεκριμένο dataset οι ταξινομητές KNN και SVM και σχολιάστηκαν τα αντιστοίχα αποτελέσματα τους. Πέραν αυτών που παρουσιάστηκαν έγινε δοκιμή(μετά την παράδοση της εργασίας) και για άλλες παραμέτρους οι οποίες οδηγούσαν σε πολυ μεγάλη διάρκεια εκτέλεσης του προγράμματος. Όπως φάνηκε οι συγκεκριμένες παράμετροι οδήγησαν σε αύξηση 15-20% του accuracy.

K-Nearest Neighbors	Support Vector Machines
Vocabulary Size = 1000	Vocabulary Size = 1000
K = 100	Kernel = Inter
 Results_KNN_1000_K=100.json - Notepad File Edit Format View Help { "Total Results": { "Correct Predictions": "1546", "Total Images": "2149", "Accuracy": "71.94043%" } }	 Results_SVM_1000_kernel_INTER.json - Notepad File Edit Format View Help { "Total Results": { "Correct Predictions": "1649", "Total Images": "2149", "Accuracy": "76.73336%" } }

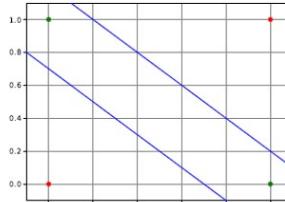
Οπως παρατηρούμε τα αποτελέσματα των συγκεκριμένων ταξινομητών έχουν αποτελέσματα κοντά στο 75%. Η προσέγγιση που θα υλοποιηθεί στην συγκεκριμένη εργασία αναμένεται να τα ξεπεράσει κατά πολύ. Την τελευταία δεκαετία και ειδικότερα μετά το 2011-2012 τα νευρωνικά δίκτυα(Neural Networks) και η βαθιά μάθηση(Deep learning) οδήγησαν σε τεράστια άλματα σε πληθώρα πεδίων και εφαρμογών, μεταξύ άλλων και της υπολογιστικής όρασης. Σε αυτό το σημείο κρίνεται απαραίτητη μία σύντομη ιστορική αναδρομή.



Παρά την τεράστια πρόοδο τα τελευταία χρόνια, οι βασικές ιδέες του deep learning που αποτελεί εξέλιξη των νευρωνικών δικτύων εισήχθησαν πολύ παλαιότερα. Ειδικότερα το 1943 οι Walter Pitts και Warren McCulloch περιέγραψαν την thresholded logic unit, μία μονάδα η οποία σχεδιάστηκε ώστε να μιμείται τον τρόπο με τον οποίο πίστευαν πως δουλεύουν οι νευρώνες. Η αναλογία μεταξύ βιολογικού και τεχνιτού νευρώνα φαίνεται παρακάτω.

Threshold Function	Biological - Artificial Neuron
$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$ 	 

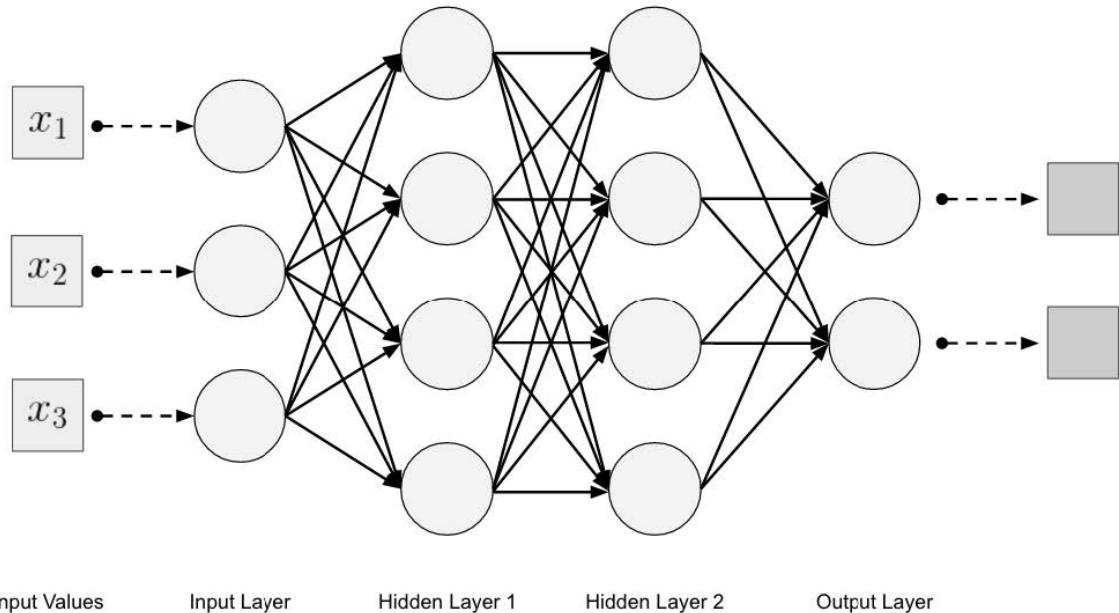
Στα τέλη της δεκαετίας του 50, ο Frank Rosenblatt και πολλοί άλλοι ερευνητές ανέπτυξαν το perceptron. Σε αντιδιαστολή με τον τεχνητό νευρώνα όπου οι παράμετροι του έπρεπε να σχεδιαστούν καθώς δεν υπήρχε κάποια μέθοδος εκπαίδευσης, ο Rosenblatt απέδειξε ότι ο κανόνας εκπαίδευσης(learning rule) που πρότεινε θα συγκλίνει στα σωστά βάρη που λύνουν το προβλήμα, εάν αυτά υπάρχουν. Το 1969 ωστόσο οι Marvin Minski και Seymour Papert είπαν ότι ένα Perceptron είναι αδύνατο να "μάθει" να λύνει το πρόβλημα του XOR. Ειδικότερα απέδειξαν θεωρητικά ότι είναι αδύνατο να μάθει να υλοποιεί μία τέτοια συνάρτηση όσο και αν το αφήσουμε να εκπαιδευτεί, πράγμα το οποίο είναι λογικό καθώς το Perceptron μπορεί να υλοποιήσει μόνο γραμμικές συναρτήσεις και η XOR είναι ΜΗ-γραμμική. Η λύση σε αυτό το πρόβλημα ήρθε μετά από αρκετά χρόνια.

Multi Layer Perceptron	2 decision Boundaries
	

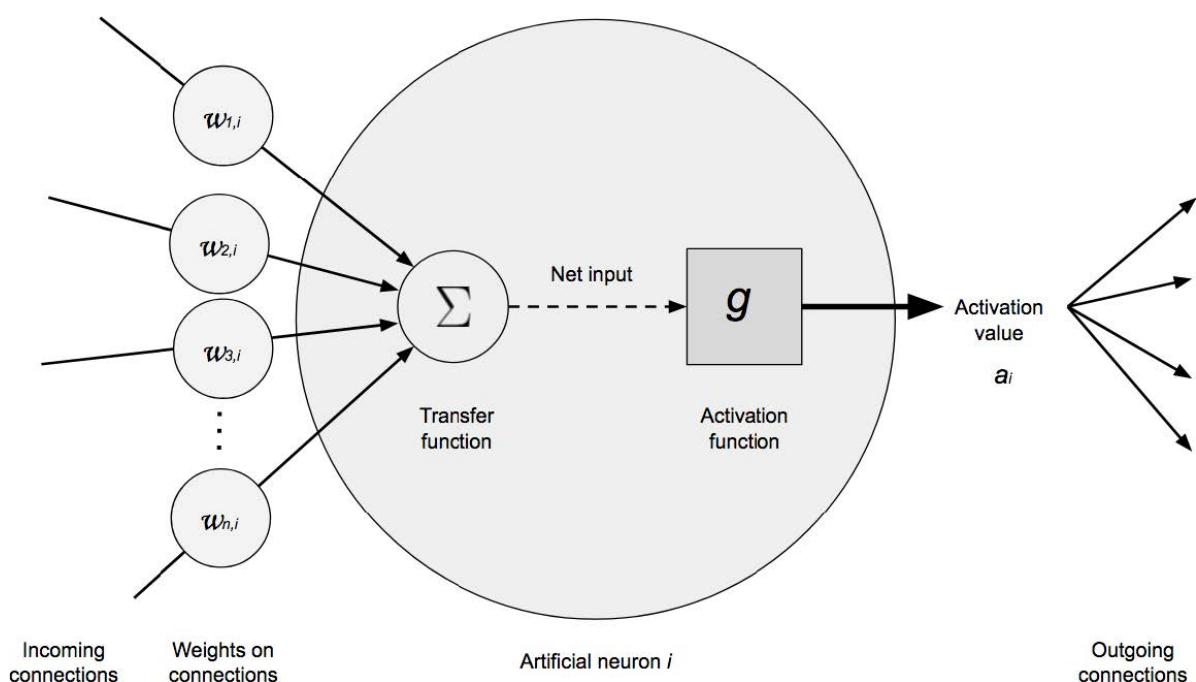
Με βάση το universal approximation theorem μπορούμε να πούμε πως ένα δίκτυο από perceptron μπορεί να χρησιμοποιηθεί για να υλοποιήσει ένα κύκλωμα-συνάρτηση που αποτελείται από πολλές πύλες NAND(κάθε πύλη NAND απαιτεί ένα υπερεπίπεδο-γραμμική συνάρτηση). Καθώς ομως οι πύλες NAND είναι καθολικές(Universal) και μπορούν να υλοποιήσουν οποιαδήποτε συνάρτηση, έτσι κατ' αντιστοιχία ένα δίκτυο πολλών(τουλάχιστον δύο) Perceptron μπορεί να προσεγγίσει και αυτό μία οποιαδήποτε συνάρτηση. Μια ακόμη σημαντική ιδέα ήταν αυτή του backpropagation. Μέσω συναρτήσεων κόστους(Cost Function) μπορεί να υπολογιστεί το loss για ένα σύνολο παραδειγμάτων στο forward propagation, και στη συνέχεια χρησιμοποιώντας κάποια μέθοδο βελτιστοποίησης μπορεί να υπολογιστεί πόσο πρέπει να αλλάξουν τα βάρη του δικτύου και να γίνει αυτή η ανανέωση σε μία διάδοση προς τα πίσω(backpropagation). Θα γίνει εκτενέστερη ανάλυση παρακάτω.

## Βαθιά Νευρωνικά Δίκτυα

Ένα πολυεπίπεδο νευρωνικό δίκτυο μπορεί να υλοποιηθεί συνενώνοντας πολλούς νευρώνες έτσι ώστε η έξοδος κάπτοι νευρώνα να αποτελεί είσοδο σε κάποιον άλλον. Συνήθως ως



Μια πολύ σημαντική προσθήκη σε αυτά που αναφέρθηκαν προηγουμένως είναι η χρήση των Activation functions. Ειδικότερα το σταθμισμένο άθροισμα των εισόδων συν κάποια σταθερά μεροληψίας(bias) που είδαμε ότι υπολογίζει ένας νευρώνας εφαρμόζεται σε μία μη γραμμική συνάρτηση και ως έξοδος πλέον του νευρώνα θεωρείται η έξοδος αυτής της συνάρτησης. Οι μη γραμμικές αυτές συναρτήσεις βοηθάνε ώστε η εκπαίδευση να γίνει σωστά.



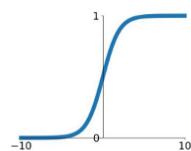
## Συναρτήσεις Ενεργοποίησης (Activation Functions)

Υπάρχει μία πληθώρα συναρτήσεων ενεργοποίησης μεταξύ των οποίων ο πιο διαδεδομένες είναι οι εξής:

### Activation Functions

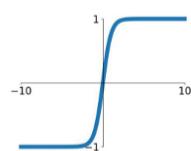
#### Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



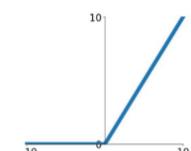
#### tanh

$$\tanh(x)$$



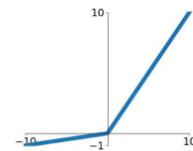
#### ReLU

$$\max(0, x)$$



#### Leaky ReLU

$$\max(0.1x, x)$$

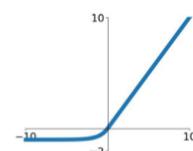


#### Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

#### ELU

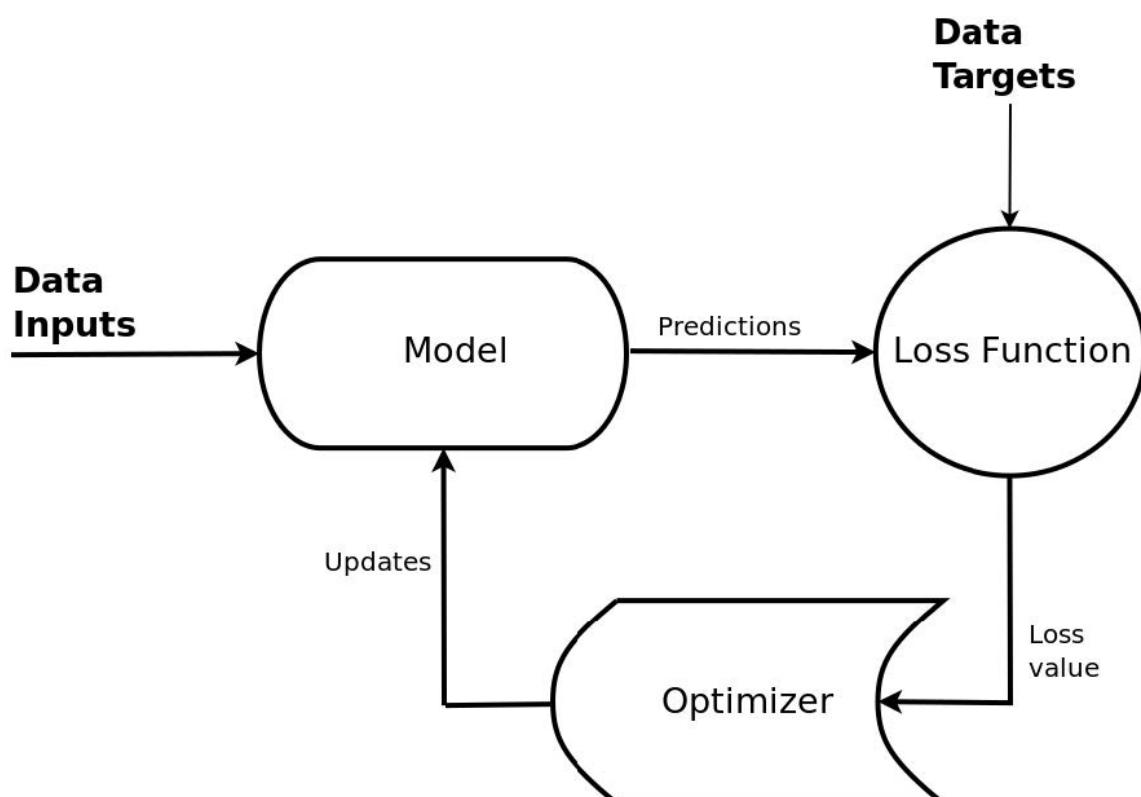
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Μεταξύ αυτών η ευρύτερα χρησιμοποιούμενη είναι η ReLU(Rectified Linear Unit) καθώς και άλλες παραλλαγές της που καλούνται να λύσουν προβλήματα όπως τω

### Επιβλεπόμενη μάθηση(Supervised Learning)

Με τον όρο μάθηση στα Νευρωνικά δίκτυα αναφερόμαστε στην αλλαγή των βαρών των συνδέσεων μεταξύ των νευρώνων με στόχο την καλύτερη ακρίβεια του δικτύου. Ένας τρόπος μάθησης είναι η επιβλεπόμενη μάθηση. Η διαδικασία που ακολουθείται στη συγκεκριμένη περίπτωση συνοψίζεται με το παρακάτω σχήμα.



Ειδικότερα, ένα μοντέλο παίρνει ένα σύνολο εισόδων, κάνει κάποιες προβλέψεις, τις συγκρίνει με τις πραγματικές τιμές των εξόδων, υπολογίζεται ποσοτικά το σφάλμα μέσω κάποιου loss function και μετά με χρήση κάποιου optimizer που έχει ως στόχο την ελαχιστοποίηση του σφάλματος αυτού ανανεώνονται τα βάρη (με βάση κάποιον κανόνα που ορίζει ο optimizer) και με τη χρήση του backpropagation.

Η διαδικασία αυτή πραγματοποιείται πολλές φορές ώστε να φτάσουμε σε βέλτιστο αποτέλεσμα. Πριν περάσουμε στην ανάλυση για τους optimizers και τα loss functions είναι σημαντικό να γινει αναφορά στις έννοιες epoch και batch. Ως **epoch(εποχή)** αναφερόμαστε σε έναν κύκλο όπου όλα τα δεδομένα έχουν περάσει μέσα στο δίκτυο μια φορά για forward propagation και μία για backpropagation. Ως **batch(παρτίδα, ομάδα)** αναφερόμαστε σε ένα μέρος των δεδομένων ου εισάγονται και επεξεργάζονται ταυτόχρονα στο δίκτυο. Συνεπώς η ανανέωση των βαρών γίνεται μετά από κάθε batch.

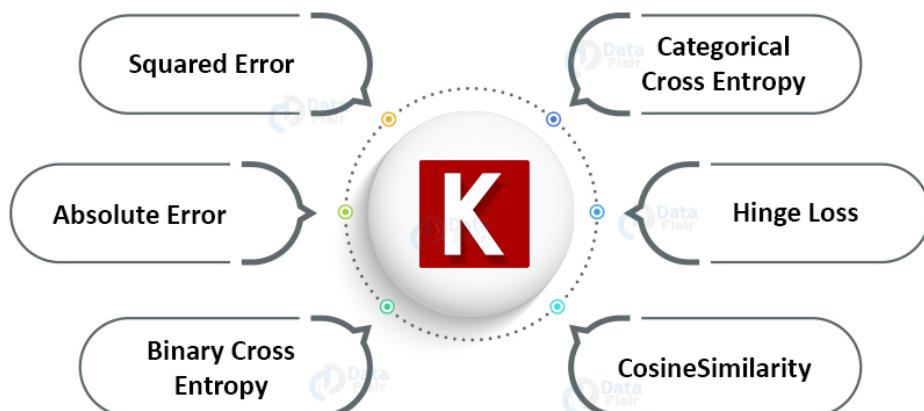
## Optimizers - Loss/Cost Functions

Το πιο σημαντικό κομμάτι στην εκπαίδευση είναι η ανανέωση των βαρών. Ο τρόπος με τον οποίο θα ανανεώνονται τα βάρη υπαγορεύεται από τον optimizer. Ο πιο γνωστός και ταυτόχρονα απλός optimizer είναι η Gradient Descend. Συγκεκριμένα η ανανέωση των βαρών γίνεται με βάση των εξής τύπο:  $\theta = \theta - \alpha \cdot \nabla J(\theta)$  όπου  $J(\theta)$  το σφάλμα που υπολογίζεται μέσω του cost function. Ωστόσο, η ανάγκη για αντιμετώπιση προβλημάτων και μεγαλύτερη ταχύτητα σύγκλισης οδήγησαν στον να σχεδιαστούν νέοι optimizers. Μερικοί γνωστοί optimizers βασισμένοι στον Gradient Descent είναι:

- Stochastic Gradient Descent
- Mini-Batch Gradient Descent(Improvement on both Stochastic Gradient Descent and standard Gradient Descent )
- Momentum -> Update Rule:  $\theta = \theta - V(t)$  όπου  $V(t) = \gamma V(t-1) + \alpha \cdot \nabla J(\theta)$
- Adagrad -> This optimizer changes the learning rate
- AdaDelta ->It is an extension of **AdaGrad** which tends to remove the *decaying learning Rate* problem of it.
- Adam(Adaptive Moment Estimation)

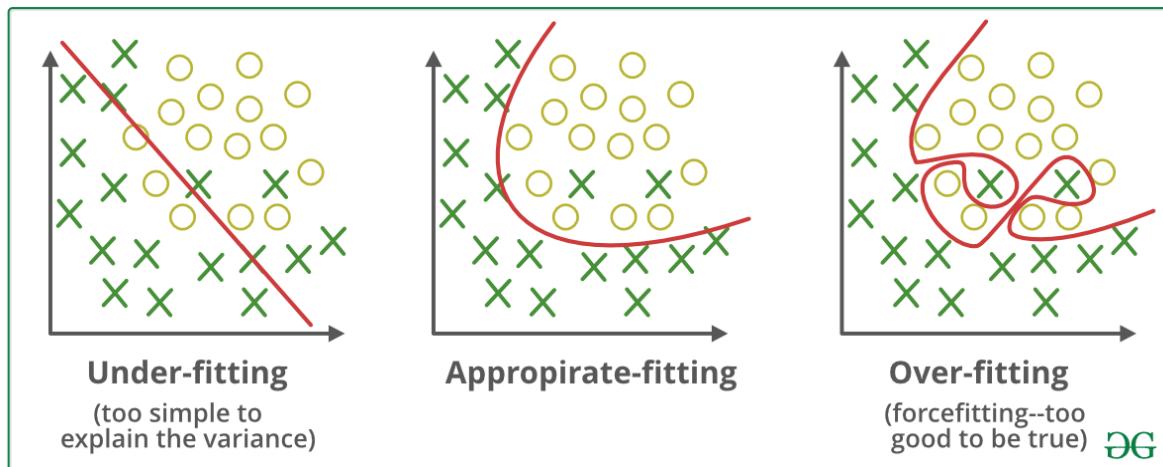
Αξίζει ιδιαίτερα να τονιστεί ότι οι optimizers βασίζονται στην παράγωγο του cost function (το οποίο επηρεάζεται σημαντικά και από το είδος της activation function) καθώς και σε μία παράμετρο η οποία λέγεται Learning rate και συμβολίζεται με το  $\alpha$  στον παραπάνω τύπο. Μέσω του learning rate ουσιαστικά ορίζουμε το πόσο μεγάλα βήματα θα κάνουμε και συνεπώς πόσο πολύ θα αλλάξουν τα βάρη. Ειδικότερα πολύ μικρή learning rate μπορεί να οδηγήσει σε τοπικά ελάχιστα που δεν είναι ικανοποιητικά ενώ πολύ μεγάλα βήματα μπορεί να αποτρέψουν και τη συγκλιση. Συνεπώς, η επιλογή του απαιτεί προσοχή. Οσον αφορά τα Cost Functions η επιλογή τους βασίζεται κυρίως στον τύπο δεδομένων και προβλήματος προς επίλυση.

## Common Loss & Functions

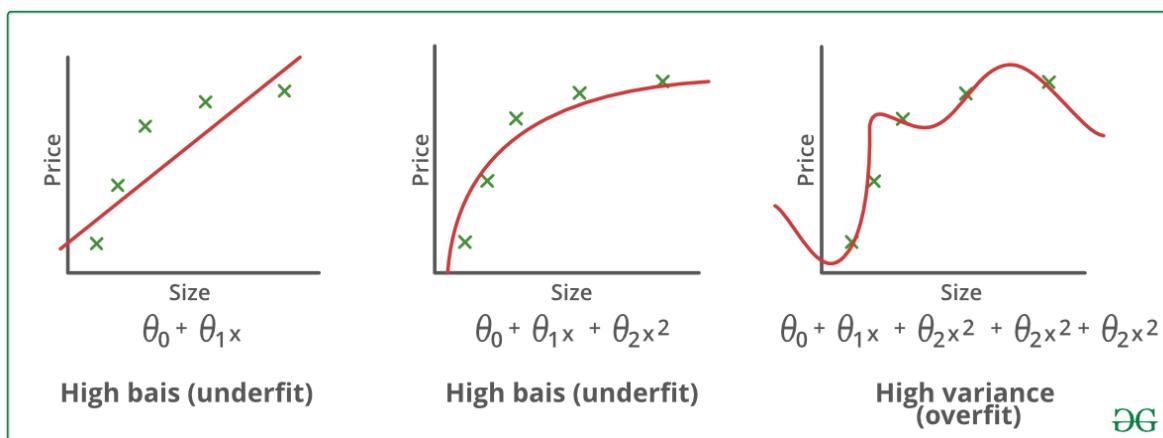


## Overfitting - Underfitting

Δύο πολύ σημαντικοί όροι που σχετίζονται με το αποτέλεσμα της εκπαίδευσης είναι η κατάσταση overfit(υπερπροσαρμογής) και underfit(υποπροσαρμογή). Με τον όρο **overfit** εννούμε ότι το μοντέλο αρχίζει και μαθαίνει πολύ καλά τα δεδούντα εκπαίδευσης αλλά χάνει την δυνατότητα να γενικεύει με αποτέλεσμα η απόδοση σε καινούργια δεδούντα που δεν έχει "δει" το δίκτυο να είναι πολύ χαμηλή. Αντίθετα με τον όρο **underfit** εννούμε ότι το μοντέλο δεν έχει μάθει σημαντικά χαρακτηριστικά των δεδούντων και έχει παραμείνει αρκετά απλό.



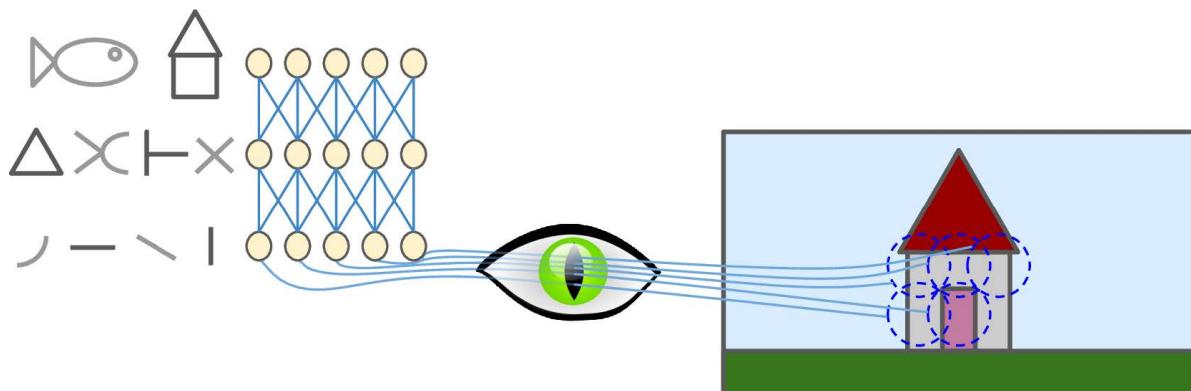
Στα παραπάνω σχήματα φαίνονται ξεκάθαρα οι 3 περιπτώσεις δηλαδή η περίπτωση του overfit, του underfit και η περίπτωση που το μοντέλο έχει κάνει την κατάλληλη προσαρμογή. Στις 2 ακραίες περιπτώσεις είναι αρκετά εύκολο να βρεθεί ένα μοντέλο. Σε κατάσταση overfit από την μία μπορεί να καταλήξουμε σε περίπτωση που ένα πολύ μεγάλο και σύνθετο μοντέλο εκπαιδευται σε ένα πολύ dataset, εάν δεν εφαρμόζεται κάποια μέθοδος regularization και πιθανώς εαν αφήσουμε την εκπαίδευση να γίνει για πάρα πολλές εποχές. Από την άλλη σε κατάσταση underfit μπορεί να βρεθούμε εαν το dataset είναι αρκετά μεγάλο και συνθετο και το μοντέλο είναι σχετικά απλό ή εάν δεν εκπαιδεύτηκε αρκετά. Στην βιβλιογραφία ακόμη εκτός των όρων underfit και overfit θα συναντήσουμε και τους όρους bias και variance. Αυτό συμβαίνει καθώς στην κατάσταση του underfit παρατηρείται αυξημένη τιμή του bias δηλαδή σχετικά μεγάλο training error και αντιστοιχης τάξης validation error ενώ σε κατάσταση overfit παρατηρείται μεγάλη τιμή στη variance (απόκλιση) δηλαδή ένα σχετικά μικρό training error αλλά ενα σημαντικά μεγαλύτερο(πολλαπλάσιο) validation error. Στην περίπτωση του μεγάλου bias δηλαδή έχουμε ένα μοντέλο που αδυνατεί να μάθει βασικά χαρακτηριστικά των δεδούντων ενώ στην περίπτωση της μεγάλης variance το μοντέλο προσαρμόζεται ακόμη και σε μή χρησιμα χαρακτηριστικά η ακόμη και το θόρυβο. Αντίστοιχα, τα διαγράμματα με το bias / variance trade-off φαίνονται παρακάτω.



# Συνελικτικά Νευρωνικά Δίκτυα (Convolutional Neural Networks)

Όπως είδαμε προηγουμένως ένας νευρώνας συνδέεται με όλους τους νευρώνες του επόμενου επιπέδου. Αυτό σημαίνει ότι μία τέτοια αρχιτεκτόνική θα ήταν απαγορευτική για μεγάλες εικόνες καθώς θα είχαμε υπορβολικά μεγάλο αριθμό παραμέτρων. Ακόμη, για να τροφοδοτήσουμε σε ένα κλασικό νευρωνικό δίκτυο μια εικόνα πρέπει πρώτα να την μετατρέψουμε σε ένα διάνυσμα. Έτσι, καθώς το νευρωνικό δεν έχει κάποια γνώση για το πώς οργανώνονται τα pixel και χάνεται η χωρική πληροφορία.

Μελέτες στον οπτικό φλοιό (visual cortex) του ανθρώπου έδειξαν ότι πολλοί νευρώνες έχουν ένα τοπικό πεδίο αντίληψης, δηλαδή ανταποκρίνονται σε μια συγκεκριμένη περιοχή του οπτικού πεδίου. Ακόμη, αυτά τα τοπικά πεδία αντίληψης μπορούν να αλληλεπικαλύπτονται και συνολικά καλύπτουν ολόκληρο το οπτικό πεδίο. Ακόμη, κάποιοι νευρώνες έχουν μεγαλύτερα πεδία αντίληψης και αντιδρούν σε πιο σύνθετα σχήματα/μοτίβα/πρότυπα (patterns). Έτσι, προήλθε η ιδέα ότι η ιδέα ότι σε μία εικόνα αντιλαμβανόμαστε πρώτα low-level χαρακτηριστικά όπως ακμές, γραμμές και απλά σχήματα τα οποία δίνονται ως σε υψηλότερου επιπέδου νευρώνες οι οποίοι μαθαίνουν να αναγνωρίζουν πιο σύνθετα γνωρίσματα. Πολύ σημαντικά ορόσημα στην ιστορία των συνελικτικών δικτύων είναι το Neocognitron το 1980 και το μοντέλο LeNet που περιγράφεται στο paper των Yann LeCun, Leon Bottou, Yoshua Bengio το 1998 και το AlexNet το 2012.



Στην παραπάνω εικόνα βλεπουμε αυτό που περιγράφηκε προς τα πάνω, δηλαδή ότι από κάτω προς τα πάνω αντιλαμβανόμαστε όλο και πιο σύνθετα γνωρίσματα. Παίρνοντας υπόψην τα παραπάνω, τα συνελικτικά νευρωνικά βασίζονται σε 3 βασικές ιδέες:

1. **Local receptive fields** (τοπικά πεδία αντίληψης)
2. **Shared weights** (κοινά βάροη)
3. **Pooling** (υποδειγματοληψία).

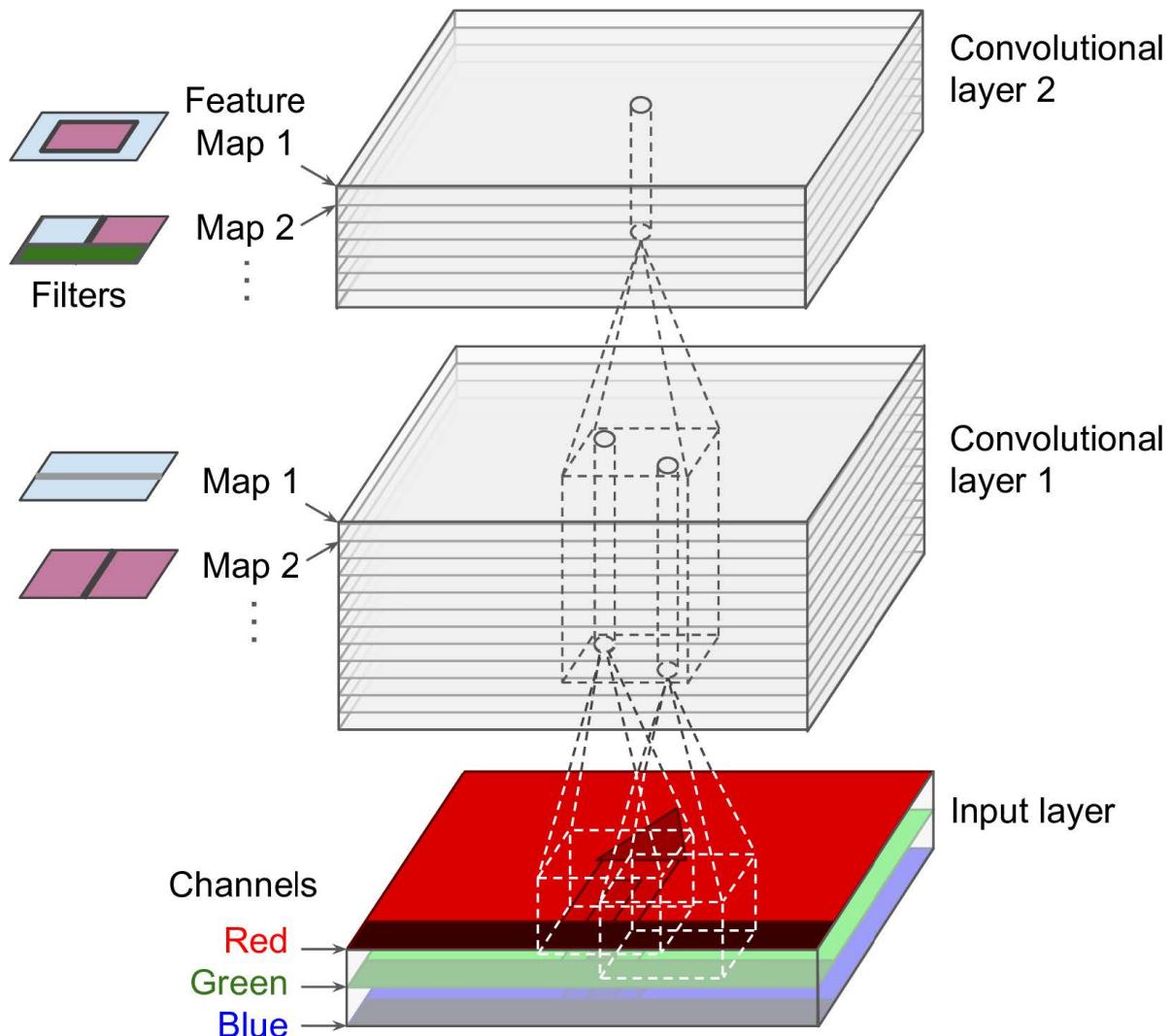
Ως τοπικό πεδίο αντίληψης ενός νευρώνα αναφερόμαστε σε αυτό το παραλληλόγραμμο-τετράγωνο στο παραπάνω σχήμα. Εδικότερα, σε μία εικόνα μέσω της πράξης της συνέλιξης εφαρμόζονται φίλτρα. Για κάθε τοπικό πεδίο αντίληψης ένας νευρώνας μαθαίνει ένα βάρος και ενα bias. Καθώς το φίλτρο μετακινείται πάνω σε μία εικόνα κατα ένα βήμα ίσο με το stride length. Μέσω των κοινών βαρών και του bias ορίζεται το φίλτρο το οποίο διατρέχει όλη την εικόνα και παράγει αυτό που ονομάζουμε feature map. Εκτός από τη συνέλιξη χρησιμοποιείται και pooling δηλαδή υποδειγματοληψία. Το pooling χρησιμοποιείται μετά τα επίπεδα συνέλιξης και έχει ώστε να συγκεντρώσει/συμπυκνώσει πληροφορία πριν περάσει στο επόμενο επίπεδο συνέλιξης. Συνηθισμένοι τρόποι για pooling είναι το max pooling και το average pooling. Παρακάτω παρουσιάζονται σε σχήματα όσα ήδαμε παραπάνω. Όσον αφορά την πράξη της συνέλιξης ισχυεί ο παρακάτω τύπος.

*Equation 14-1. Computing the output of a neuron in a convolutional layer*

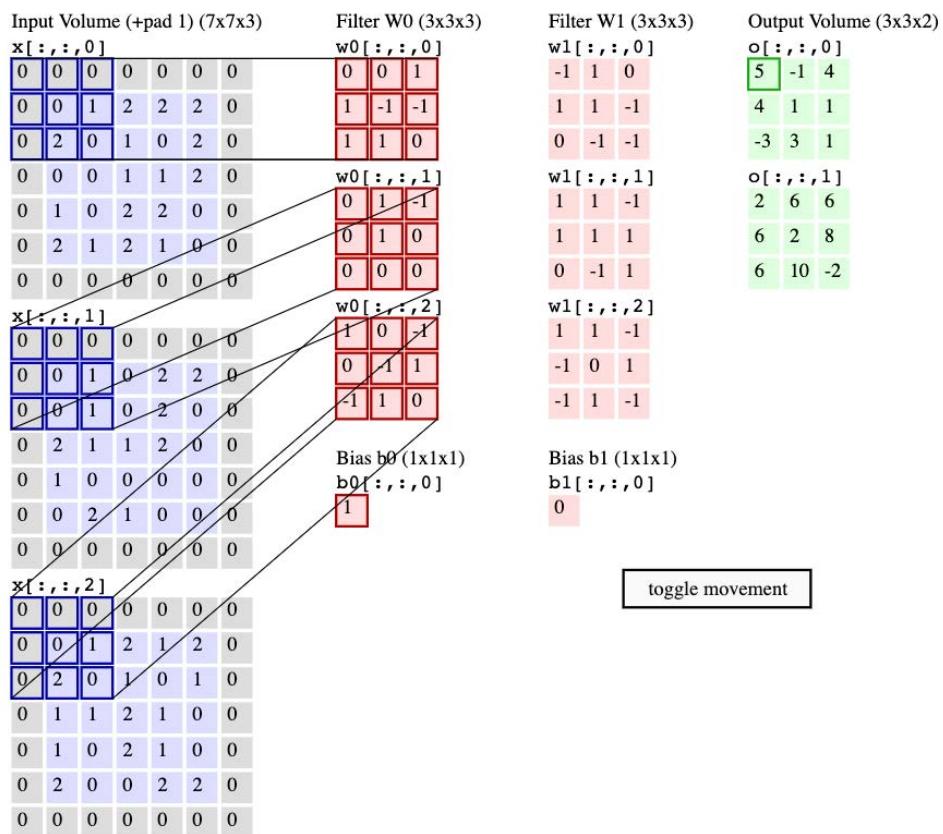
$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with } \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

- $z_{i,j,k}$  is the output of the neuron located in row  $i$ , column  $j$  in feature map  $k$  of the convolutional layer (layer  $l$ ).
- As explained earlier,  $s_h$  and  $s_w$  are the vertical and horizontal strides,  $f_h$  and  $f_w$  are the height and width of the receptive field, and  $f_{n'}$  is the number of feature maps in the previous layer (layer  $l - 1$ ).
- $x_{i',j',k'}$  is the output of the neuron located in layer  $l - 1$ , row  $i'$ , column  $j'$ , feature map  $k'$  (or channel  $k'$  if the previous layer is the input layer).
- $b_k$  is the bias term for feature map  $k$  (in layer  $l$ ). You can think of it as a knob that tweaks the overall brightness of the feature map  $k$ .
- $w_{u,v,k',k}$  is the connection weight between any neuron in feature map  $k$  of the layer  $l$  and its input located at row  $u$ , column  $v$  (relative to the neuron's receptive field), and feature map  $k'$ .

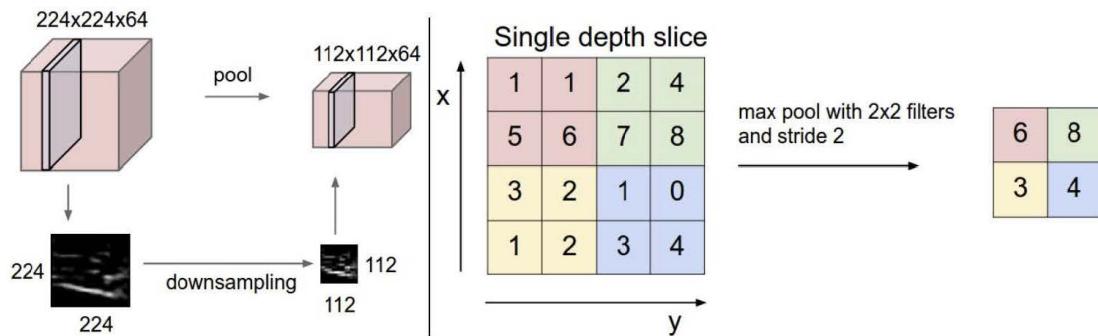
Εδώ παρατηρούμε τα επίπεδα συνέλιξης τα φίλτρα και τα feature maps που παράγονται.



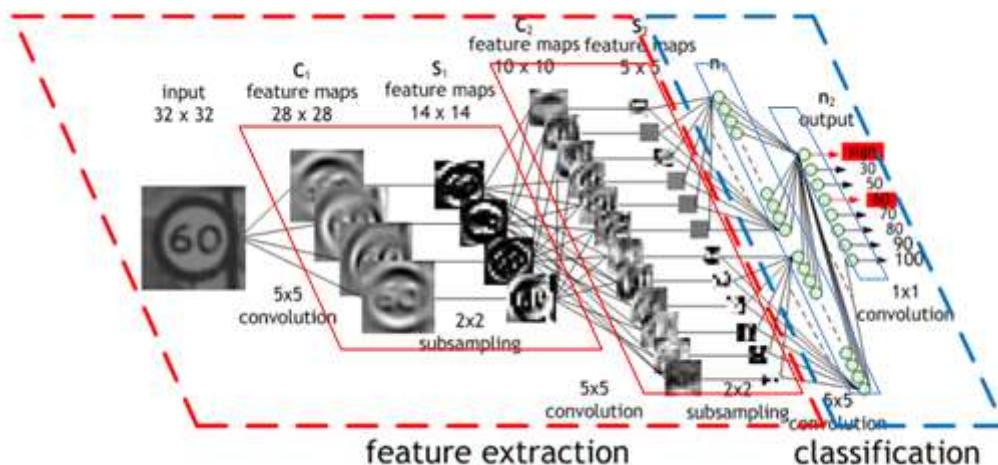
Η συνέλιξη σχηματικά φαίνεται ως εξής:



Αντίστοιχα η πράξη του max-pooling:



Τέλος, αξίζει να αναφέρουμε ότι μια σειρά convolution και pooling απαιτείται ένα τελικό κομμάτι με dense νευρώνες το οποίο λειτουργεί ως classifier.



# Κώδικας - Προγραμματιστική υλοποίηση

Πριν γίνει ανάλυση των 2 υλοποιήσεων καλο είναι να γίνει αναφορά στην δομή των αρχείων. Υπάρχουν 3 notebooks. Ένα για το μή-προεκπαιδευμένο δίκτυο, ένα για το προεκπαιδευμένο και ένα που χρησιμοποιήθηκε για φόρτωση των μοντέλων και για διάφορες οπτικοποιήσεις. Ακόμη, αξίζει να αναφερθεί πως το dataset που δόθηκε δεν περιέχει validation set. Το validation set ωστόσο είναι απαραίτητο για την εκπαίδευση του δικτύου καθώς αποτελείται από δεδομένα που δεν χρησιμοποιούνται για εκπαίδευση αλλά για έλεγχο κατά την διάρκεια της εκπαίδευσης. Συνεπώς, μέσω αυτού μπορούν να ρυθμιστούν οι διάφοροι υπερπαράμετροι του δικτύου ενώ ακόμη μπορούν να εξαχθουν συμπεράσματα τόσο για την ακρίβεια όσο και για το αν εμφανίζεται το φαινόμενο της υπερπροσαρμογής(overfit). Έτσι, λοιπόν για να ξεπεραστεί αυτό το πρόβλημα έγινε χρήση της συνάρτησης `ImageDataGenerator` μέσω του ορίσματος `validation split` και στη συνέχεια στην συνάρτηση `flow_from_directory` ρυθμίζοντας κατάλληλα το όρισμα `subset` και κρατώντας σταθερό `seed`. Ειδικότερα έγινε διάσπαση του training set σε 80-20 για training και validation. Δημιουργήθηκε τεχνητά δηλαδή ένα validation set. Για να φαίνονται και τα αποτελέσματα στο notebook θα παρουσιαστούν με μορφή screenshot και όχι code snippet.

## Μη-Προεκπαιδευμένο Δίκτυο

Αρχικά, γίνεται `import` των απαραίτητων βιβλιοθηκών καθώς και του `drive`. Ακόμη, σε περίπτωση που δεν είναι φορτωμένα τα δεδομένα στο colab τα φορτώνουμε από το `drive` και κάνουμε `unzip` και ορίζουμε τα κατάλληλα path σε μεταβλητές.

The screenshot shows a Jupyter Notebook interface with several code cells and their outputs:

- Importing Libraries:**

```
[ ] from keras import models  
from keras import layers  
from keras import optimizers  
from keras import regularizers  
import tensorflow as tf  
import os  
import zipfile
```
- Importing Drive:**

```
[ ] from google.colab import drive  
drive.mount('/content/drive')
```

Output: Mounted at /content/drive
- Importing and unzip data from drive:**

```
[ ] #Comment the following lines if imagedb and imagedb_test are already in content folder or specify path for data in drive  
local_zip = '/content/drive/MyDrive/Colab Notebooks/CV_2020-2021_DL/HW4/imagedb_btsd.zip'  
zip_ref = zipfile.ZipFile(local_zip, 'r')  
zip_ref.extractall('/content')  
zip_ref.close()
```
- Specify train and test path-directory:**

```
[ ] train_dir = "/content/imagedb"  
test_dir = "/content/imagedb_test"
```

Στην συνέχεια πριν προχωρήσουμε στην ανάλυση του δικτύου, θα ορίσουμε 2 custom activation functions. Ειδικότερα ορίστηκαν οι συναρτήσεις ενεργοποίησης `mish` και `swish`. Μετά από δοκιμές κατέληξα πως η `mish` (η οποία θα περιγραφε στην συνέχεια) είχε τα καλύτερα αποτελέσματα και από την `swish` και την `ReLU`. Να σημειωθεί πως τόσο η `mish` όσο και η `swish` είναι παραλλαγές της `ReLU`. Τόσο για την `swish` όσο και για την `mish` παρατίθενται τα paper τα οποία της αναλύουν καθώς επίσης παρακάτω παραθέτω και 2 εικόνες από τα loss Landscape που παράγεται και συγκρίνονται `ReLU`, `Swish`, `Mish`. Αξίζει, ωστόσο να σημειωθεί πως μία πιο έμπιστη μέθοδος για να επικυρώσουμε πως όντως αποδίδει καλύτερα η `mish` είναι αυτή του K - Fold Cross Validation. Ωστόσο, στα πλαίσια της συγκεκριμένης εργασίας δεν χρησιμοποιήθηκε η συγκεκριμένη μέθοδος.

## Costum implementation for Swish activation function

Official Paper: <https://arxiv.org/pdf/1710.05941v1.pdf>

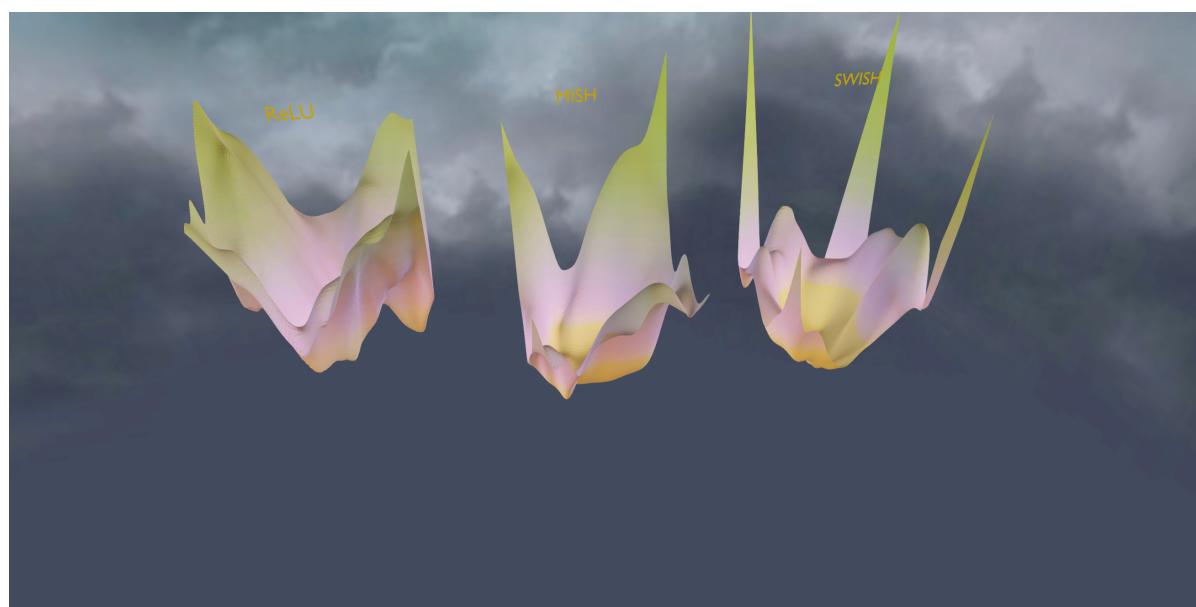
```
[ ] from keras.backend import sigmoid
def swish(x, beta = 1):
    return (x * sigmoid(beta * x))
from keras.utils.generic_utils import get_custom_objects
from keras.layers import Activation
get_custom_objects().update({'swish': swish})
# Actually it is already implemented in tensorflow -> tf.nn.swish
```

## Custom implementation for Mish activation function

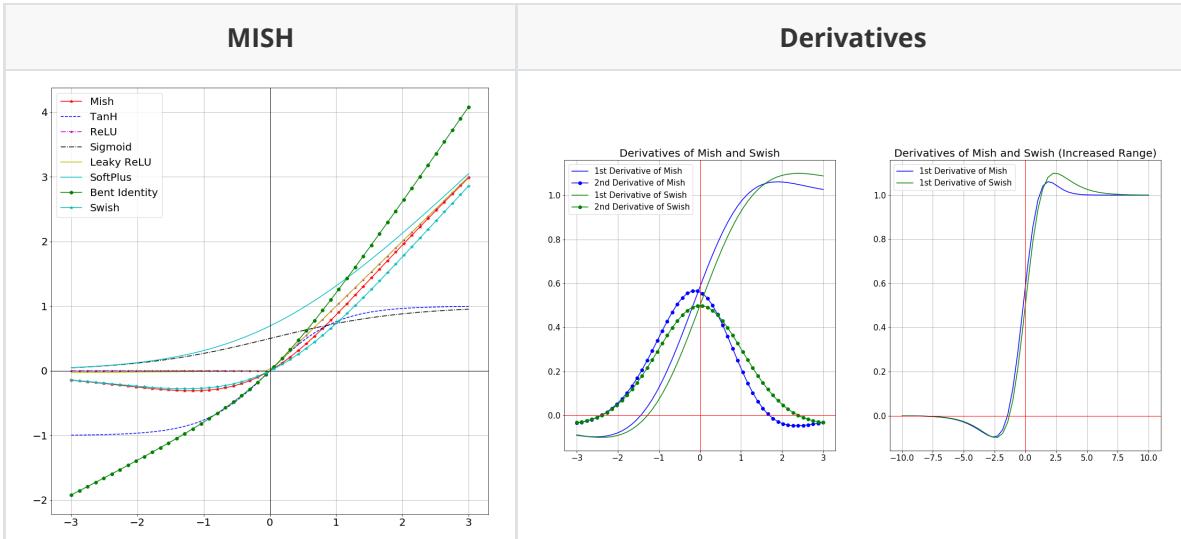
Official Paper: <https://arxiv.org/pdf/1908.08681.pdf>

```
➊ from keras.backend import tanh,softplus
def mish(x):
    return x * tanh(softplus(x))
from keras.utils.generic_utils import get_custom_objects
from keras.layers import Activation
get_custom_objects().update({'mish':mish})
```

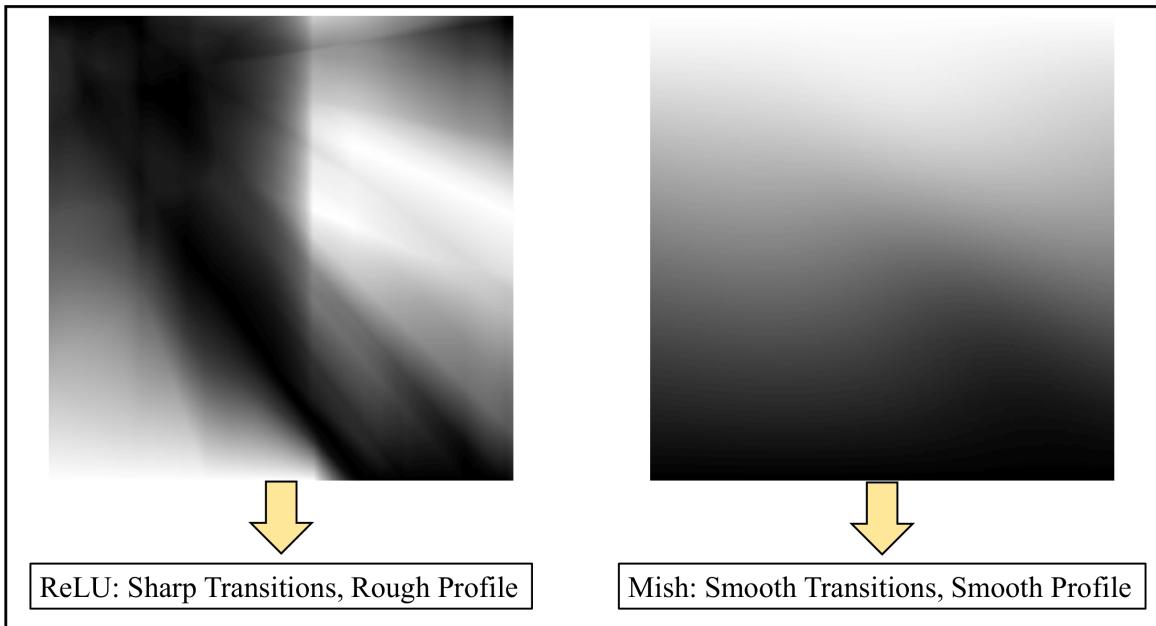
Παρά το γεγονός ότι η swish υλοποιήθηκε και custom, παρατήρησα τελικά πως υπάρχει έτοιμη υλοποίηση στο tensorflow. Εδώ αξίζει να δούμε και τις διαφορές στα loss landscape (Έχουν παραχθεί με το δίκτυο ResNet - 20 για το dataset CIFAR 10 για 200 εποχές).



Όπως βλέπουμε και στον κώδικα η mish ορίζεται ως εξής:  $f(x) = x \cdot \tanh \cdot (\text{softpuls}(x))$



Σε γενικές γραμμές η mish παρέχει καλύτερη ακρίβεια, μικρότερο συνολικό loss και ένα πιο λείο και πιο εύκολα στο να βελτιστοποιηθεί loss-landscape σε σύγκριση με Swish και ReLU.



Στην συνέχεια προχωράμε με την περιγραφή του μοντέλου. Πέραν την χρήση της mish ως activation function αξίζει να δόθει έμφαση στα εξής: Όσο προχωράμε στο δίκτυο ο αριθμός των φίλτρων αυξάνεται. Αρχικά 16, στη συνέχεια 32 και μετά 64. Αυτή η αύξηση έχει νόημα καθώς ο αριθμός των βασικών και χαμηλού επιπέδου χαρακτηριστικών(ακμές,κάποια πολύ βασικά σχήματα,μικροί κύκλοι) είναι περιορισμένα αλλά καθώς βαθαίνει το δίκτυο υπάρχουν πολλοί τρόποι με τους οποίους μπορούν αυτά τα χαρακτηριστικά να συνδιαστούν και να παράγουν υψηλού επιπέδου χαρακτηριστικά. Ειδικότερα είναι συνηθισμένη πρακτική ο αριθμός των φίλτρων να διπλασιάζεται μετά από κάθε pooling layer. Αυτό συμβαίνει καθώς κάθε pooling layer διαιρεί τις χωρικές διαστάσεις δια δύο και συνεπώς μπορούμε να διπλασιάσουμε τον αριθμό των φίλτρων/kernel χωρίς να υπάρχει ανησυχία για τεράστια άυξηση στις παραμέτρους, στη χρήση μνήμης και στο υπολογιστικό φορτίο. Ακόμη, παρατηρούμε ότι οι εικόνες που δέχεται το δίκτυο είναι 128 \* 128 και έχουν 3 κανάλια. Στο dataset το μέγεθος των εικόνων είναι μεταβλητό αλλά μέσω των generator στην συνέχεια το ρυθμίζουμε και αυτό. Επιπλέον, χρησιμοποιείται 2 φορές το μοτίβο conv->conv -> pooling ώστε να διατηρούνται οι χωρικές διαστάσεις των feature maps.

Model	Summary
<p>Model Description</p> <pre>#Create the model model = models.Sequential() model.add(layers.Conv2D(16, (7, 7),activation="mish", padding="same", input_shape=(128,128 ,3))) model.add(layers.BatchNormalization()) model.add(layers.MaxPooling2D(pool_size=(2, 2),strides=(2,2))) model.add(layers.Conv2D(32, (5, 5),activation="mish", padding="same" )) model.add(layers.BatchNormalization()) model.add(layers.Conv2D(32, (5, 5),activation="mish", padding="same" )) model.add(layers.BatchNormalization()) model.add(layers.MaxPooling2D(pool_size=(2, 2),strides=(2,2))) model.add(layers.Conv2D(64, (3, 3),activation="mish", padding="same" )) model.add(layers.BatchNormalization()) model.add(layers.Conv2D(64, (3, 3),activation="mish", padding="same" )) model.add(layers.BatchNormalization()) model.add(layers.MaxPooling2D(pool_size=(2, 2).strides=(2,2))) model.add(layers.Flatten()) model.add(layers.Dense(512,activation="mish")) model.add(layers.BatchNormalization()) model.add(layers.Dropout(0.3)) model.add(layers.Dense(256,activation="mish")) model.add(layers.BatchNormalization()) model.add(layers.Dropout(0.25)) model.add(layers.Dense(128,activation="mish")) model.add(layers.BatchNormalization()) model.add(layers.Dropout(0.2)) model.add(layers.Dense(34,activation="softmax"))  model.summary()</pre>	<pre>Model: "sequential_14" Layer (type)          Output Shape         Param # conv2d_66 (Conv2D)    (None, 128, 128, 16)      2368 batch_normalization_67 (Batch Normalization) (None, 64, 64, 16)      64 max_pooling2d_39 (MaxPooling) (None, 64, 64, 16)      0 conv2d_67 (Conv2D)    (None, 64, 64, 32)      12832 batch_normalization_106 (Batch Normalization) (None, 64, 64, 32)      128 conv2d_68 (Conv2D)    (None, 64, 64, 32)      25632 batch_normalization_107 (Batch Normalization) (None, 64, 64, 32)      128 max_pooling2d_40 (MaxPooling) (None, 32, 32, 32)      0 conv2d_69 (Conv2D)    (None, 32, 32, 64)      18496 batch_normalization_108 (Batch Normalization) (None, 32, 32, 64)      256 conv2d_70 (Conv2D)    (None, 32, 32, 64)      36928 batch_normalization_109 (Batch Normalization) (None, 32, 32, 64)      256 max_pooling2d_41 (MaxPooling) (None, 16, 16, 64)      0 flatten_13 (Flatten)  (None, 16384)      0 dense_52 (Dense)     (None, 512)      8389120 batch_normalization_110 (Batch Normalization) (None, 512)      2048 dropout_39 (Dropout) (None, 512)      0 dense_53 (Dense)     (None, 256)      131328 batch_normalization_111 (Batch Normalization) (None, 256)      1024 dropout_40 (Dropout) (None, 256)      0 dense_54 (Dense)     (None, 128)      32896 batch_normalization_112 (Batch Normalization) (None, 128)      512 dropout_41 (Dropout) (None, 128)      0 dense_55 (Dense)     (None, 34)      4386 Total params: 8,658,462 Trainable params: 8,656,194 Non-trainable params: 2,208</pre>

Οπως βλέπουμε έγινε χρήση και dropout καθώς και batch normalization. Αυτές είναι δύο τεχνικές για regularization ώστε να αποφευχθεί το overfitting. Στο όρισμα των dropout ορίζουμε το ποσοστό των νευρώνων το οποίο θα αγνοηθεί κατά την διάρκεια της εκπαίδευσης σε ένα step αλλα μπορεί να είναι ενεργοποιημένοι στο επόμενο step. Το batch normalization από την άλλη κάνει κανονικοποίηση στην ενεργοποίηση σε κάθε παρτίδα(batch) και συγκεκριμένα εφαρμόζει μετασχηματισμούς ώστε ο μέσος όρος να είναι κοντά στο 0 ενώ η απόκλιση να είναι κοντά στο 1.

Στην συνέχεια περνάμε στους Data Generators

Data Generators
<pre>from tensorflow.keras.preprocessing.image import ImageDataGenerator  train_datagen = ImageDataGenerator(rescale=1./255,horizontal_flip=True,vertical_flip=True,                                     zoom_range=0.2,rotation_range=20,                                     featurewise_center=True,                                     featurewise_std_normalization=True,                                     validation_split=0.2)  val_datagen = ImageDataGenerator(rescale=1./255, validation_split=0.2)  # ----- # Flow training images in batches of  using train_datagen generator # ----- train_generator = train_datagen.flow_from_directory(train_dir,   batch_size=32,   class_mode='categorical',   # color_mode='grayscale',   target_size=(128,128),   shuffle=True,   subset='training', seed=1)  # ----- # Flow validation images in batches of  using test_datagen generator # ----- validation_generator =  train_datagen.flow_from_directory(train_dir,   batch_size=32,   class_mode='categorical',   # color_mode='grayscale',   target_size=(128,128),   subset='validation', seed=1)  Found 2457 images belonging to 34 classes. Found 599 images belonging to 34 classes.</pre>

Συγκεκριμένα, χρησιμοποιήθηκε και Data Augmentation(horizontal\_flip=True,vertical\_flip=True, zoom\_range=0.2,rotation\_range=20, featurewise\_center=True, featurewise\_std\_normalization=True). Ετσι, τα δεδομένα εκπαίδευσης δίνονται με κάποια πιθανότητα μετασχηματισμένα στο δίκτυο έτσι ώστε αυτό να μάθει να γενικεύει καλύτερα. Αξίζει να σημειωθεί ότι τα δεδομένα αυτά δεν προστίθενται στα ήδη υπάρχοντα αλλά οι μετασχηματισμοί γίνονται live στην εκπαίδευση και οι μετασχηματισμένες εικόνες δίνονται στο δίκτυο αντί αυτών που υπάρχουν. Επιπλέον πρέπει να πούμε ότι augmentation έχει νόημα να γίνει στο train και όχι στο test set και validation set καθώς μετά χάνεται το νόημα για το οποίο το χρησιμοποιούμε. Τέλος παρατηρούμε αυτό που αναφέρθηκε στην αρχή για τον χωρισμό του training σε 2 subsets. Το μέγεθος των εικόνων ορίζεται στο 128 \* 128 και το batch size στο 32 καθώς μετά από αναζήτηση βρήκα πως σχετικά μικρό batch size όπως η τιμή 32(για τέτοιου μεγέθους εικόνες) φαινεται να βελτιώνει την ικανότητα του δικτύου να γενικεύει και να έρχεται σε σύγκλιση. Η παρατήρηση μου αυτή βασίζεται στο παρακάτω paper. <https://arxiv.org/pdf/1804.07612.pdf>

Στην συνέχεια περνάμε στην εκπαίδευση του μοντέλου. Πριν από ορίζουμε και callbacks. Μέσω αυτών μπορούμε να κάνουμε έλεγχο κατά την διάρκεια του training. Συγκεκριμένα χρησιμοποίησα ModelCheckpoint έτσι ώστε να γίνεται save το μοντέλο σε συγκεκριμένες εποχές. By default γίνεται έλεγχος στο val\_loss και με βάση εάν αυτό βελτιώνεται τότε αποθηκεύουμε το μοντέλο με τα τρέχον βάρη του. Ακόμη, μέσω του EarlyStoping ορίζουμε τον αριθμό των εποχών στις οποίες θα σταματήσει η εκπαίδευση εάν δεν υπάρχει βελτίωση στο val\_loss που ορίσαμε παραπάνω με στόχο να αποτρέψουμε φαινόμενα όπως το overfitting.

```
import datetime

callbacks = []

logdir = os.path.join("/content/drive/MyDrive/Colab Notebooks/CV_2020-2021_DL/Hw4/logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1)
callbacks.append(tensorboard_callback)

save_best_callback = tf.keras.callbacks.ModelCheckpoint(f"/content/drive/MyDrive/Colab Notebooks/CV_2020-2021_DL/Hw4/best_weights_mish_aug_v3.hdf5", save_best_only=True, verbose=1)
callbacks.append(save_best_callback)

early_stop_callback = tf.keras.callbacks.EarlyStopping(patience=12, restore_best_weights=True, verbose=1)
callbacks.append(early_stop_callback)

# Compile the model
model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.Adam(1e-2e-4),
              metrics=['acc'])

# Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples/train_generator.batch_size,
    epochs=150,
    validation_data=validation_generator,
    verbose=1,
    callbacks=callbacks)

# Save the model
```

Στην συνέχεια περνάμε στο κομμάτι της εκπαίδευσης. Όπως παρατηρούμε αρχικά πως ξεκινάμε από που χαμηλά accuracy και υψηλά loss. Μέσα σε 5 εποχές ήδη το μοντέλο έχει προσεγγίσει το 95% accuracy στο validation set.

```
/usr/local/lib/python3.6/dist-packages/keras/preprocessing/image/image_data_generator.py:720: UserWarning: This ImageDataGenerator specifies 'featurewise_center', but it hasn't been fit on any training data. Fit it first by calling
  warnings.warn("This ImageDataGenerator specifies "
/usr/local/lib/python3.6/dist-packages/keras/preprocessing/image/image_data_generator.py:728: UserWarning: This ImageDataGenerator specifies 'featurewise_std_normalization', but it hasn't been fit on any training data. Fit it first
  warnings.warn("This ImageDataGenerator specifies "
Epoch 1/150
76/76 [=====] - 14s 16ms/step - loss: 3.0283 - acc: 0.2534 - val_loss: 3.0106 - val_acc: 0.0768

Epoch 00001: val_loss improved from Inf to 3.61062, saving model to /content/drive/MyDrive/Colab Notebooks/CV_2020-2021_DL/Hw4/best_weights_mish_aug_v3.hdf5
Epoch 2/150
76/76 [=====] - 12s 156ms/step - loss: 1.4263 - acc: 0.6074 - val_loss: 3.1688 - val_acc: 0.2220

Epoch 00002: val_loss improved from 3.61062 to 3.16884, saving model to /content/drive/MyDrive/Colab Notebooks/CV_2020-2021_DL/Hw4/best_weights_mish_aug_v3.hdf5
Epoch 3/150
76/76 [=====] - 12s 156ms/step - loss: 0.9833 - acc: 0.7326 - val_loss: 2.3899 - val_acc: 0.3689

Epoch 00003: val_loss improved from 3.16884 to 2.38986, saving model to /content/drive/MyDrive/Colab Notebooks/CV_2020-2021_DL/Hw4/best_weights_mish_aug_v3.hdf5
Epoch 4/150
76/76 [=====] - 12s 155ms/step - loss: 0.7084 - acc: 0.8144 - val_loss: 1.4329 - val_acc: 0.6327

Epoch 00004: val_loss improved from 2.38986 to 1.43286, saving model to /content/drive/MyDrive/Colab Notebooks/CV_2020-2021_DL/Hw4/best_weights_mish_aug_v3.hdf5
Epoch 5/150
76/76 [=====] - 12s 157ms/step - loss: 0.5300 - acc: 0.8576 - val_loss: 1.0008 - val_acc: 0.7412

Epoch 00005: val_loss improved from 1.43286 to 1.00078, saving model to /content/drive/MyDrive/Colab Notebooks/CV_2020-2021_DL/Hw4/best_weights_mish_aug_v3.hdf5
Epoch 6/150
76/76 [=====] - 12s 156ms/step - loss: 0.4799 - acc: 0.8795 - val_loss: 0.5554 - val_acc: 0.8447

Epoch 00006: val_loss improved from 1.00078 to 0.55535, saving model to /content/drive/MyDrive/Colab Notebooks/CV_2020-2021_DL/Hw4/best_weights_mish_aug_v3.hdf5
Epoch 7/150
76/76 [=====] - 12s 153ms/step - loss: 0.4101 - acc: 0.9000 - val_loss: 0.3776 - val_acc: 0.8965

Epoch 00007: val_loss improved from 0.55535 to 0.37761, saving model to /content/drive/MyDrive/Colab Notebooks/CV_2020-2021_DL/Hw4/best_weights_mish_aug_v3.hdf5
Epoch 8/150
76/76 [=====] - 12s 156ms/step - loss: 0.3520 - acc: 0.9057 - val_loss: 0.2969 - val_acc: 0.9115

Epoch 00008: val_loss improved from 0.37761 to 0.29690, saving model to /content/drive/MyDrive/Colab Notebooks/CV_2020-2021_DL/Hw4/best_weights_mish_aug_v3.hdf5
Epoch 9/150
76/76 [=====] - 12s 156ms/step - loss: 0.3078 - acc: 0.9230 - val_loss: 0.2953 - val_acc: 0.9082

Epoch 00009: val_loss improved From 0.29690 to 0.29531, saving model to /content/drive/MyDrive/Colab Notebooks/CV_2020-2021_DL/Hw4/best_weights_mish_aug_v3.hdf5
Epoch 10/150
76/76 [=====] - 12s 154ms/step - loss: 0.2271 - acc: 0.9298 - val_loss: 0.2316 - val_acc: 0.9249

Epoch 00010: val_loss improved From 0.29531 to 0.23160, saving model to /content/drive/MyDrive/Colab Notebooks/CV_2020-2021_DL/Hw4/best_weights_mish_aug_v3.hdf5
Epoch 11/150
76/76 [=====] - 12s 154ms/step - loss: 0.2051 - acc: 0.9372 - val_loss: 0.1947 - val_acc: 0.9499
```

## Αποτελέσματα

Η εκπαίδευση τελικά σταμάτησε στις 47 εποχές. Ειδικότερα αποθηκεύτηκε το μοντέλο της εποχής 36(35 στο tensorboard) με val\_loss =0.1075 και v\_acc=97,16%

```
Epoch 00035: val loss did not improve from 0.11301
Epoch 36/150
76/76 [=====] - 12s 154ms/step - loss: 0.0616 - acc: 0.9833 - val_loss: 0.1075 - val_acc: 0.9716
Epoch 00036: val_loss improved from 0.11301 to 0.10750, saving model to /content/drive/MyDrive/Colab_Notebooks/CV_2020-2021_DL/HW4/best_weights_mish_aug_v3.hdf5
Epoch 37/150
76/76 [=====] - 12s 157ms/step - loss: 0.0643 - acc: 0.9828 - val_loss: 0.1413 - val_acc: 0.9616
Epoch 00037: val_loss did not improve from 0.10750
Epoch 38/150
76/76 [=====] - 12s 152ms/step - loss: 0.0580 - acc: 0.9851 - val_loss: 0.1575 - val_acc: 0.9633
Epoch 00038: val_loss did not improve from 0.10750
Epoch 39/150
76/76 [=====] - 12s 153ms/step - loss: 0.0556 - acc: 0.9859 - val_loss: 0.1828 - val_acc: 0.9549
Epoch 00039: val_loss did not improve from 0.10750
Epoch 40/150
76/76 [=====] - 12s 153ms/step - loss: 0.0607 - acc: 0.9835 - val_loss: 0.2074 - val_acc: 0.9282
Epoch 00040: val_loss did not improve from 0.10750
Epoch 41/150
76/76 [=====] - 12s 153ms/step - loss: 0.0468 - acc: 0.9870 - val_loss: 0.1533 - val_acc: 0.9616
Epoch 00041: val_loss did not improve from 0.10750
Epoch 42/150
76/76 [=====] - 12s 154ms/step - loss: 0.0597 - acc: 0.9815 - val_loss: 0.1395 - val_acc: 0.9583
Epoch 00042: val_loss did not improve from 0.10750
Epoch 43/150
76/76 [=====] - 12s 153ms/step - loss: 0.0594 - acc: 0.9825 - val_loss: 0.1232 - val_acc: 0.9699
Epoch 00043: val_loss did not improve from 0.10750
Epoch 44/150
76/76 [=====] - 12s 153ms/step - loss: 0.0487 - acc: 0.9892 - val_loss: 0.1113 - val_acc: 0.9716
Epoch 00044: val_loss did not improve from 0.10750
Epoch 45/150
76/76 [=====] - 12s 153ms/step - loss: 0.0354 - acc: 0.9915 - val_loss: 0.1368 - val_acc: 0.9549
Epoch 00045: val_loss did not improve from 0.10750
Epoch 46/150
76/76 [=====] - 12s 157ms/step - loss: 0.0478 - acc: 0.9842 - val_loss: 0.1262 - val_acc: 0.9699
Epoch 00046: val_loss did not improve from 0.10750
Epoch 47/150
76/76 [=====] - 12s 156ms/step - loss: 0.0418 - acc: 0.9871 - val_loss: 0.2204 - val_acc: 0.9149
Epoch 00047: val_loss did not improve from 0.10750
Epoch 48/150
76/76 [=====] - 12s 154ms/step - loss: 0.0491 - acc: 0.9872 - val_loss: 0.1237 - val_acc: 0.9666
Epoch 00048: val_loss did not improve from 0.10750
Restoring model weights from the end of the best epoch.
Epoch 00048: early stopping
```

Στα δεδομένα ελέγχου(test set) τα αντίστοιχα αποτελέσματα είναι:

### Model Evaluation

```
▶ test_datagen = ImageDataGenerator(rescale=1./255)
# -----
# Flow validation images in batches of 20 using test_datagen generator
# -----
test_generator = test_datagen.flow_from_directory(test_dir,
                                                 batch_size=64,
                                                 class_mode='categorical',
#                                                 color_mode='grayscale',
                                                 target_size=(128,128))

loss, acc = model.evaluate(test_generator)

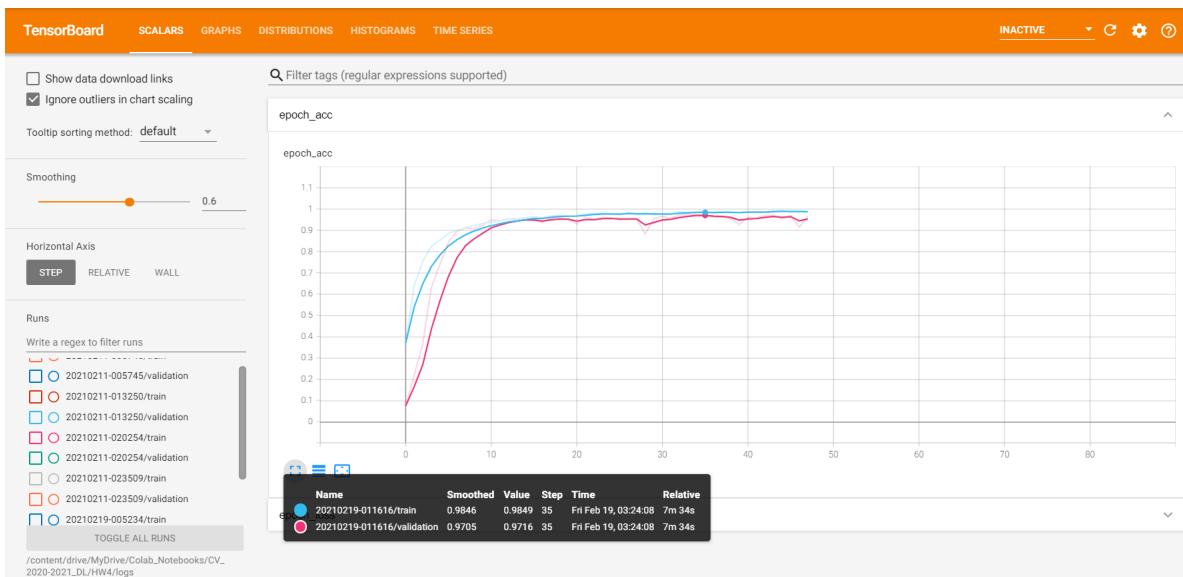
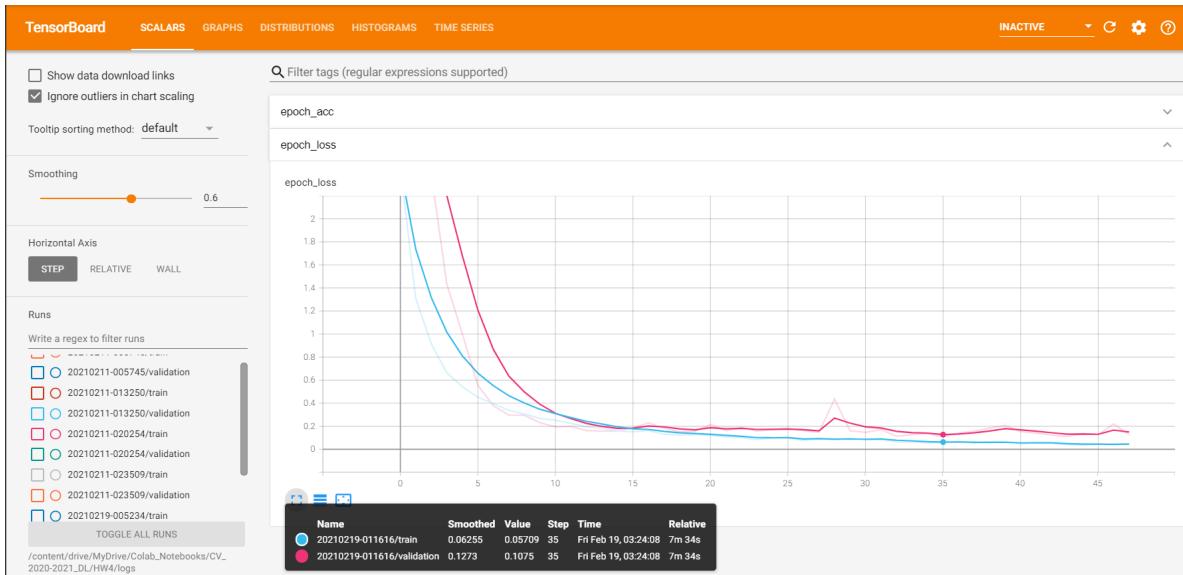
▷ Found 2149 images belonging to 34 classes.
34/34 [=====] - 1s 31ms/step - loss: 0.1321 - acc: 0.9744
```

Το συγκεκριμένο αρχείο είναι το HW4.ipynb. Κάτι τελευταίο στο οποίο δεν έγινε αναφορά πιο πριν είναι η χρήση του tensorboard. Μέσω αυτού έγιναν plot τα accuracy και τα loss για διαφόρα μοντέλα που δοκίμασα. Μέσω αυτού είδα την επιρροή των διαφόρων υπερπαραμέτρων στο δίκτυο και έκανα τις αντίστοιχες δοκιμές.

```
▶ # Load the TensorBoard notebook extension
%load_ext tensorboard

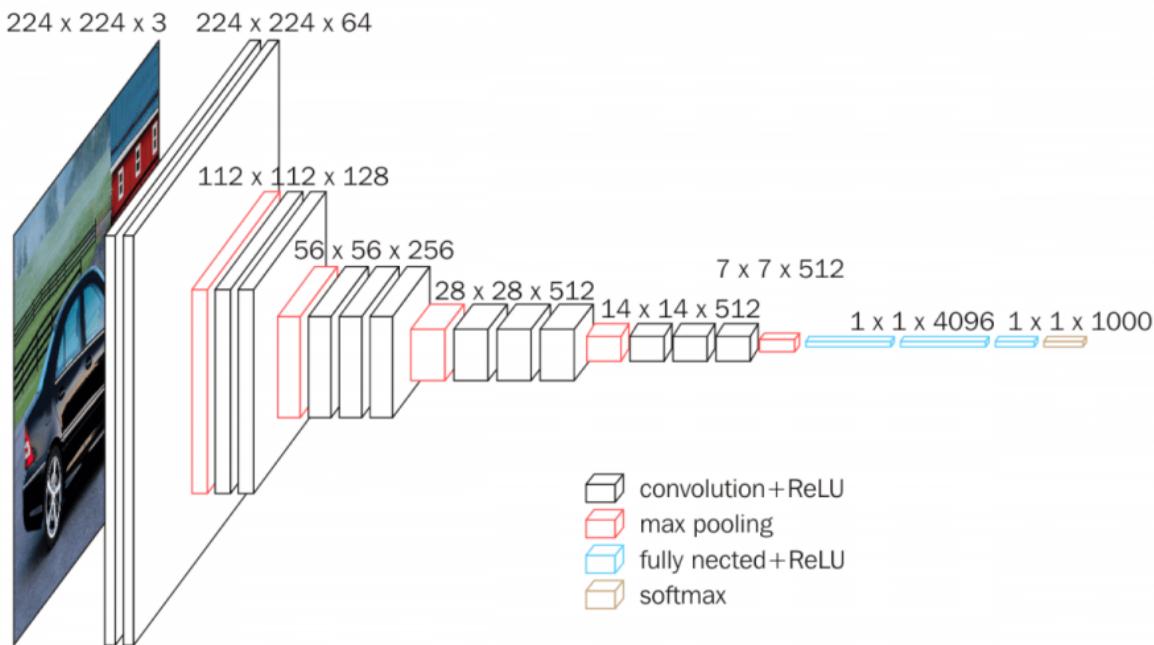
▷ The tensorboard extension is already loaded. To reload it, use:
%reload_ext tensorboard

▶ %tensorboard --logdir "/content/drive/MyDrive/Colab_Notebooks/CV_2020-2021_DL/HW4/logs"
```



## Προεκπαιδευμένο Δίκτυο

Στα πλαίσια της συγκεκριμένης εργασίας έγινε χρήση του VGG16. Τα προεκπαιδευμένα δίκτυα δηλαδή η χρήση του Transfer Learning είναι πολύ διαδεδομένη και σε πολλές εφαρμογές παρουσιάζουν πολύ καλά αποτελέσματα. Συγκεκριμένα έγινε χρήση του VGG16 καθώς αποτελείται αμιγώς από στρώματα συνέλιξης και pooling. Αντίθετα δίκτυα όπως το ResNet έχουν και άλλες συνδέσεις μεταξύ των στρωμάτων. Στην παρακάτω εικόνα παρατηρούμε τα στρώματα του VGG16.



Στην συγκεκριμένη υλοποίηση μου, φόρτωσα το VGG16 και πάγωσα όλα τα layers εκτός των 4 τελευταίων. Στην συνέχεια έβαλα και πάλι ένα δικό μου dense δίκτυο ως classifier. Αξίζει σε αυτό το σημείο να πούμε πως το VGG16 που φορτώσαμε έχει τα βάρη από εκπαίδευση στο imagenet που είναι μια τεράστια βάση εικόνων. Η επιλογή για πάγωμα των αρχικών στρωμάτων και εκπαίδευση των τελευταίων και του classifier έχει λογική εάν θυμηθούμε πως τα πρώτα στρώματα μαθαίνουν kernels/φιλτρα τα οποία ανιχνεύουν low level χαρακτηριστικά ενώ τα τελευταία στα οποία το εκπαίδεύμες ανιχνέουν high level χαρακτηριστικά. Οσον αφορά την εκπαίδευση παρατηρούμε ότι εδώ ξεκινάμε από ένα val\_accuracy κοντά στο 57% από την πρώτη κιολας εποχη σε αντίθεση με το δικό μου μοντέλο που ήταν μη προεκπαιδευμένο όπου ξεκινούσε με λιγότερο από 10%.

```
▶ from keras.applications import VGG16
  #Load the VGG model
  vgg_conv = VGG16(weights='imagenet',
                    include_top=False,
                    input_shape=(128, 128, 3))
```

```

# Freeze the layers except the last 4 layers
for layer in vgg_conv.layers[:-4]:
    layer.trainable = False

# Check the trainable status of the individual layers
for layer in vgg_conv.layers:
    print(layer, layer.trainable)

▷ <tensorflow.python.keras.engine.input_layer.InputLayer object at 0x7f5e95228cf8> False
<tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f5e9c160e10> False
<tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f5e9c160358> False
<tensorflow.python.keras.layers.pooling.MaxPooling2D object at 0x7f5e9c1d7f28> False
<tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f5e95265320> False
<tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f5e952320b8> False
<tensorflow.python.keras.layers.pooling.MaxPooling2D object at 0x7f5ece551a58> False
<tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f5ece552a58> False
<tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f5ece553780> False
<tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f5ece5539b0> False
<tensorflow.python.keras.layers.pooling.MaxPooling2D object at 0x7f5e951234a8> False
<tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f5e82633e80> False
<tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f5e803b74e0> False
<tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f5e803bd358> False
<tensorflow.python.keras.layers.pooling.MaxPooling2D object at 0x7f5e803c3860> False
<tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f5e803ca7b8> True
<tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f5e803ca1d0> True
<tensorflow.python.keras.layers.convolutional.Conv2D object at 0x7f5e803bd240> True
<tensorflow.python.keras.layers.pooling.MaxPooling2D object at 0x7f5e803d55f8> True

```

Model	Model Summary
<pre> # Create the model model = models.Sequential()  # Add the vgg convolutional base model model.add(vgg_conv) model.add(layers.Flatten()) model.add(layers.Dense(512,activation="mish")) model.add(layers.BatchNormalization()) model.add(layers.Dropout(0.3)) model.add(layers.Dense(256,activation="mish")) model.add(layers.BatchNormalization()) model.add(layers.Dropout(0.2)) model.add(layers.Dense(128,activation="mish")) model.add(layers.BatchNormalization()) model.add(layers.Dropout(0.2)) model.add(layers.Dense(34,activation="softmax"))  model.summary() </pre>	<pre> ▷ Model: "sequential" -----  Layer (type)           Output Shape        Param # vgg16 (Functional)    (None, 4, 4, 512)   14714688 flatten (Flatten)      (None, 8192)         0 dense (Dense)          (None, 512)          4194816 batch_normalization (BatchNo (None, 512)   2048 dropout (Dropout)       (None, 512)          0 dense_1 (Dense)         (None, 256)          131328 batch_normalization_1 (Batch (None, 256)   1024 dropout_1 (Dropout)     (None, 256)          0 dense_2 (Dense)         (None, 128)          32896 batch_normalization_2 (Batch (None, 128)   512 dropout_2 (Dropout)     (None, 128)          0 dense_3 (Dense)         (None, 34)           4386 -----  Total params: 19,081,698 Trainable params: 11,444,642 Non-trainable params: 7,637,056 </pre>

Στα υπόλοιπα(DataGenerators,Callbacks,ModelCheckpoint) δεν υπάρχει κάποια ιδιαίτερη διαφοροποίηση.

## Αποτελέσματα

```
Epoch 1/150
19/19 [=====] - 24s 847ms/step - loss: 3.3082 - acc: 0.2030 - val_loss: 2.2916 - val_acc: 0.5726
Epoch 00001: val_loss improved from inf to 2.29157, saving model to /content/drive/MyDrive/Colab_Notebooks/CV_2020-2021_DL/HW4/best_weights_mish_aug_pretrained.hdf5
Epoch 2/150
19/19 [=====] - 12s 636ms/step - loss: 1.3376 - acc: 0.6613 - val_loss: 1.6503 - val_acc: 0.7813
Epoch 00002: val_loss improved from 2.29157 to 1.65029, saving model to /content/drive/MyDrive/Colab_Notebooks/CV_2020-2021_DL/HW4/best_weights_mish_aug_pretrained.hdf5
Epoch 3/150
19/19 [=====] - 12s 644ms/step - loss: 0.8447 - acc: 0.8071 - val_loss: 1.2691 - val_acc: 0.8748
Epoch 00003: val_loss improved from 1.65029 to 1.26912, saving model to /content/drive/MyDrive/Colab_Notebooks/CV_2020-2021_DL/HW4/best_weights_mish_aug_pretrained.hdf5
Epoch 4/150
19/19 [=====] - 12s 639ms/step - loss: 0.5399 - acc: 0.8893 - val_loss: 0.9400 - val_acc: 0.9232
Epoch 00004: val_loss improved from 1.26912 to 0.93996, saving model to /content/drive/MyDrive/Colab_Notebooks/CV_2020-2021_DL/HW4/best_weights_mish_aug_pretrained.hdf5
Epoch 5/150
19/19 [=====] - 12s 642ms/step - loss: 0.4470 - acc: 0.9084 - val_loss: 0.6737 - val_acc: 0.9149
Epoch 00005: val_loss improved from 0.93996 to 0.67368, saving model to /content/drive/MyDrive/Colab_Notebooks/CV_2020-2021_DL/HW4/best_weights_mish_aug_pretrained.hdf5
```

Τελικά, η εκπαίδευση σταμάτησε στην 49η εποχή και αποθηκεύτηκε το μοντέλο της 39ης εποχής με val\_loss = 0.1011 και val\_acc = 97.5%

```
Epoch 00038: val loss did not improve from 0.10528
Epoch 39/150
19/19 [=====] - 12s 644ms/step - loss: 0.0597 - acc: 0.9897 - val_loss: 0.1011 - val_acc: 0.9750
Epoch 00039: val loss improved from 0.10528 to 0.10111, saving model to /content/drive/MyDrive/Colab_Notebooks/CV_2020-2021_DL/HW4/best_weights_mish_aug_pretrained.hdf5
Epoch 40/150
19/19 [=====] - 12s 639ms/step - loss: 0.0396 - acc: 0.9897 - val_loss: 0.2002 - val_acc: 0.9599
Epoch 00040: val_loss did not improve from 0.10111
Epoch 41/150
19/19 [=====] - 12s 622ms/step - loss: 0.0285 - acc: 0.9944 - val_loss: 0.1618 - val_acc: 0.9616
Epoch 00041: val_loss did not improve from 0.10111
Epoch 42/150
19/19 [=====] - 12s 622ms/step - loss: 0.0230 - acc: 0.9976 - val_loss: 0.1205 - val_acc: 0.9666
Epoch 00042: val_loss did not improve from 0.10111
Epoch 43/150
19/19 [=====] - 12s 621ms/step - loss: 0.0206 - acc: 0.9987 - val_loss: 0.1240 - val_acc: 0.9633
Epoch 00043: val_loss did not improve from 0.10111
Epoch 44/150
19/19 [=====] - 12s 649ms/step - loss: 0.0234 - acc: 0.9975 - val_loss: 0.1020 - val_acc: 0.9666
Epoch 00044: val_loss did not improve from 0.10111
Epoch 45/150
19/19 [=====] - 12s 628ms/step - loss: 0.0293 - acc: 0.9961 - val_loss: 0.1330 - val_acc: 0.9649
Epoch 00045: val_loss did not improve from 0.10111
Epoch 46/150
19/19 [=====] - 12s 622ms/step - loss: 0.0175 - acc: 0.9983 - val_loss: 0.1275 - val_acc: 0.9683
Epoch 00046: val_loss did not improve from 0.10111
Epoch 47/150
19/19 [=====] - 12s 627ms/step - loss: 0.0160 - acc: 0.9981 - val_loss: 0.1547 - val_acc: 0.9633
Epoch 00047: val_loss did not improve from 0.10111
Epoch 48/150
19/19 [=====] - 12s 619ms/step - loss: 0.0145 - acc: 0.9989 - val_loss: 0.1729 - val_acc: 0.9599
Epoch 00048: val_loss did not improve from 0.10111
Epoch 49/150
19/19 [=====] - 12s 622ms/step - loss: 0.0152 - acc: 0.9981 - val_loss: 0.1966 - val_acc: 0.9566
Epoch 00049: val_loss did not improve from 0.10111
Restoring model weights from the end of the best epoch.
Epoch 00049: early stopping
```

Όσον αφορά το test set τα αποτελέσματα είναι τα εξής:

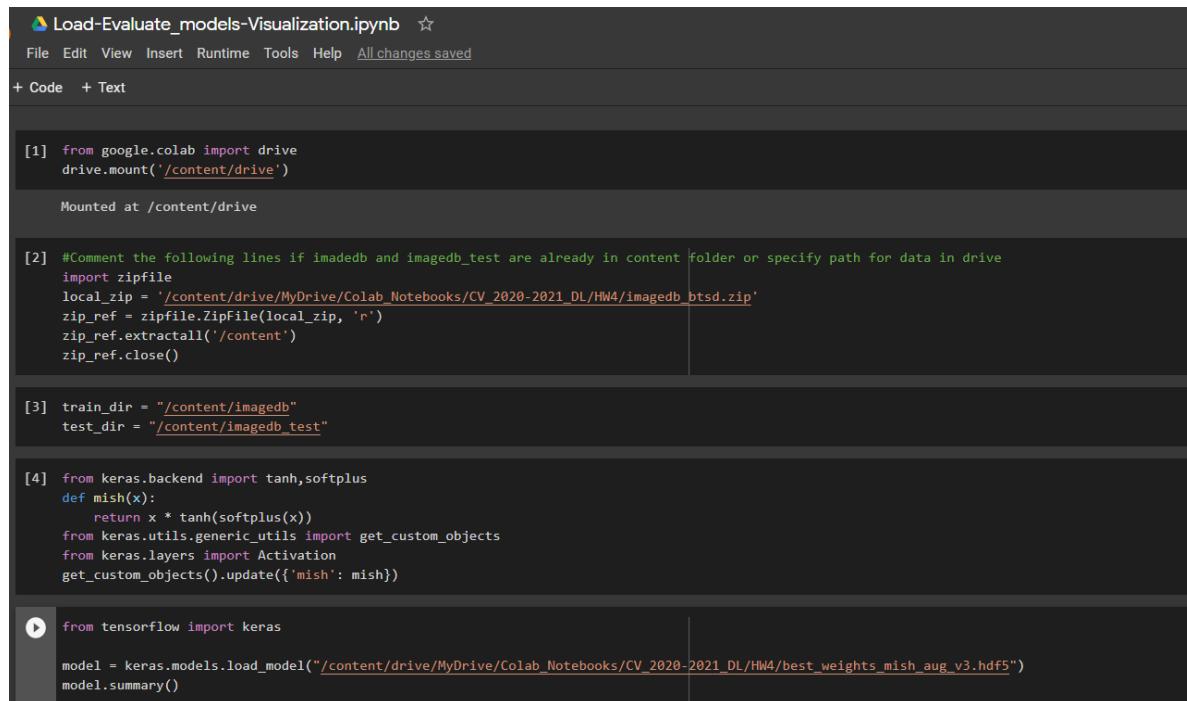
```
▶ test_datagen = ImageDataGenerator(rescale=1./255)
# -----
# Flow validation images in batches of 20 using test_datagen generator
# -----
test_generator = test_datagen.flow_from_directory(test_dir,
                                                 batch_size=64,
                                                 class_mode='categorical',
#                                                 color_mode='grayscale',
                                                 target_size=(128,128))
loss, acc = model.evaluate(test_generator)

▷ Found 2149 images belonging to 34 classes.
34/34 [=====] - 5s 113ms/step - loss: 0.1191 - acc: 0.9716
```

Παρατηρούμε ότι το Accuracy στα δεδομένα ελέγχου είναι ελάχιστα μικρότερο από του δικού μου δικτύου(από 97.44 σε 97.16). Ωστόσο, το loss ειναι ειναι αισθητά μικρότερο(από 0.1321 σε 0.1191).

## Φόρτωση Μοντέλων - Οπτικοποίηση

Σε συνέχεια των 2 notebook, δημιούργησα και ένα τρίτο ώστε να μπορώ να φορτώνω τα μοντέλα που έχω εκπαιδευσει και να δοκιμάσω κάποια visualizations.



```
[1] from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive

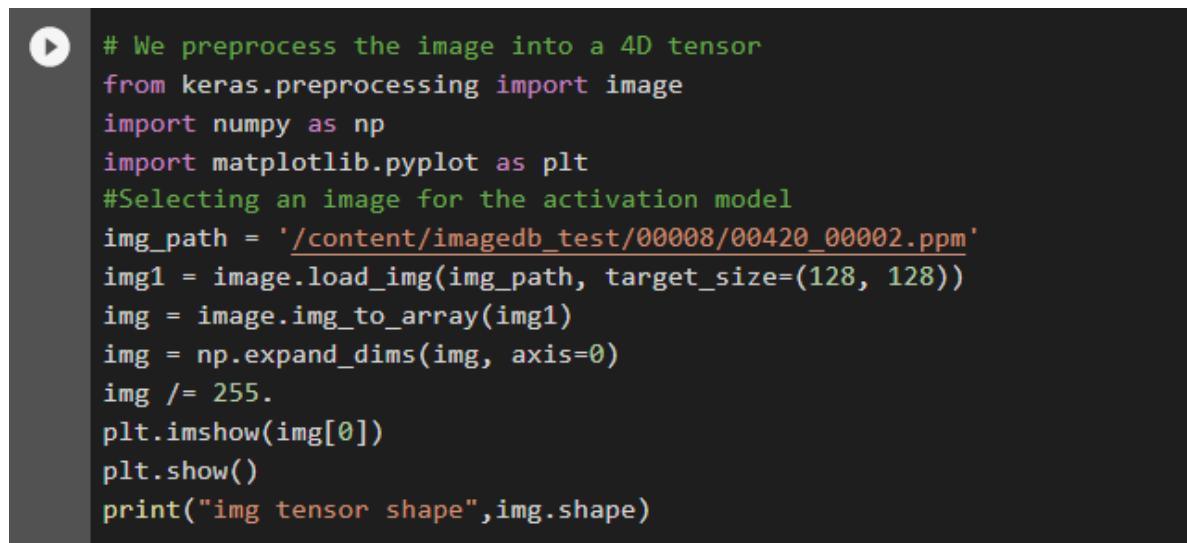
[2] #Comment the following lines if imagedb and imagedb_test are already in content folder or specify path for data in drive
import zipfile
local_zip = '/content/drive/MyDrive/Colab_Notebooks/CV_2020-2021_DL/HW4/imagedb_btsd.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('/content')
zip_ref.close()

[3] train_dir = "/content/imagedb"
test_dir = "/content/imagedb_test"

[4] from keras.backend import tanh,softplus
def mish(x):
    return x * tanh(softplus(x))
from keras.utils.generic_utils import get_custom_objects
from keras.layers import Activation
get_custom_objects().update({'mish': mish})

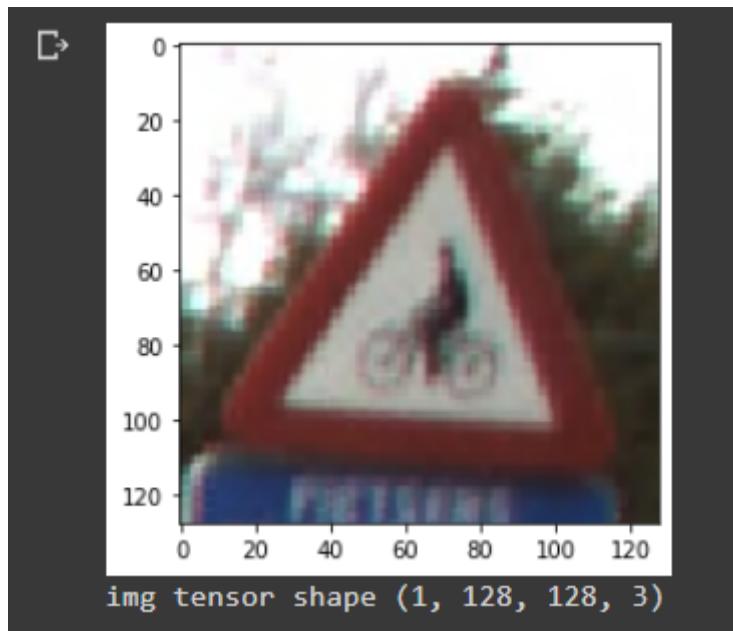
▶ from tensorflow import keras
model = keras.models.load_model("/content/drive/MyDrive/Colab_Notebooks/CV_2020-2021_DL/HW4/best_weights_mish_aug_v3.hdf5")
model.summary()
```

Οπως βλέπουμε μπορώ να φορτώσω το μοντέλο το οποίο έχω αποθηκευμένο στο drive πλέον. Σε πρώτο στάδιο μέσω μια συνάρτησης του keras μπορούμε να πάρουμε μια περιγραφή του υπάρχοντος μοντέλου σε μορφή εικόνας που μας δίνει όλες τις απαραίτητες πληροφορίες και μπορεί να χρησιμοποιηθεί για να περιγραφεί μια αρχιτεκτονική. Σε δεύτερο στάδιο βασιζόμενος στον παρακάτω κώδικα/notebook <https://colab.research.google.com/github/fchollet/deep-learning-with-python-notebooks/blob/master/5.4-visualizing-what-convnets-learn.ipynb#scrollTo=EydxebyMFb6w> κατάφερα να οπτικοποίήσω τα feature maps σε κάθε στάδιο του δικτύου μου εάν το τροφοδοτήσω με μια εικόνα πχ από το test set. Αρχικά, κάνουμε τις απαραίτητες μετατροπές.



```
# We preprocess the image into a 4D tensor
from keras.preprocessing import image
import numpy as np
import matplotlib.pyplot as plt
#Selecting an image for the activation model
img_path = '/content/imagedb_test/00008/00420_00002.ppm'
img1 = image.load_img(img_path, target_size=(128, 128))
img = image.img_to_array(img1)
img = np.expand_dims(img, axis=0)
img /= 255.
plt.imshow(img[0])
plt.show()
print("img tensor shape",img.shape)
```

Το αποτέλεσμα είναι το εξής.

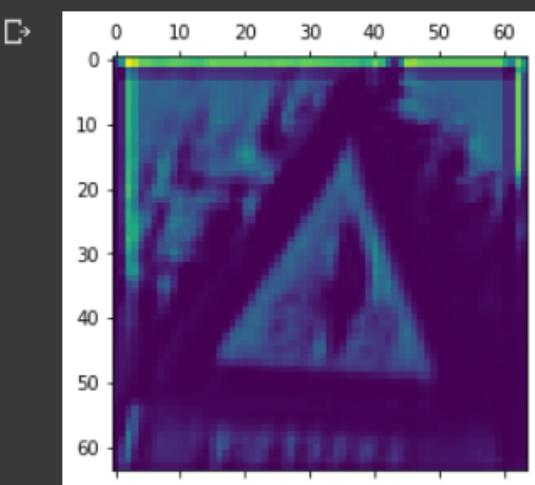


```
[ ] from keras import models

# Extracts the outputs of the top 8 layers:
layer_outputs = [layer.output for layer in model.layers[:13]]
# Creates a model that will return these outputs, given the model input:
activation_model = models.Model(inputs=model.input, outputs=layer_outputs)

[ ] # This will return a list of 5 Numpy arrays:
# one array per layer activation
activations = activation_model.predict(img)

▶ plt.matshow(activations[4][0, :, :, 31], cmap='viridis')
plt.show()
```



Εδώ παρατηρούμε το feature map του 4ου layer και του 32ου φίλτρου. Ωστόσο ιδανικά θα θέλαμε να δούμε όλα τα feature maps για όλα τα Layers. Αυτό το επιτυγχάνουμε με τον παρακάτω κώδικα.

```

# These are the names of the layers, so can have them as part of our plot
layer_names = []
for layer in model.layers[:13]:
    layer_names.append(layer.name)

images_per_row = 16

# Now let's display our feature maps
for layer_name, layer_activation in zip(layer_names, activations):
    # This is the number of features in the feature map
    n_features = layer_activation.shape[-1]

    # The feature map has shape (1, size, size, n_features)
    size = layer_activation.shape[1]

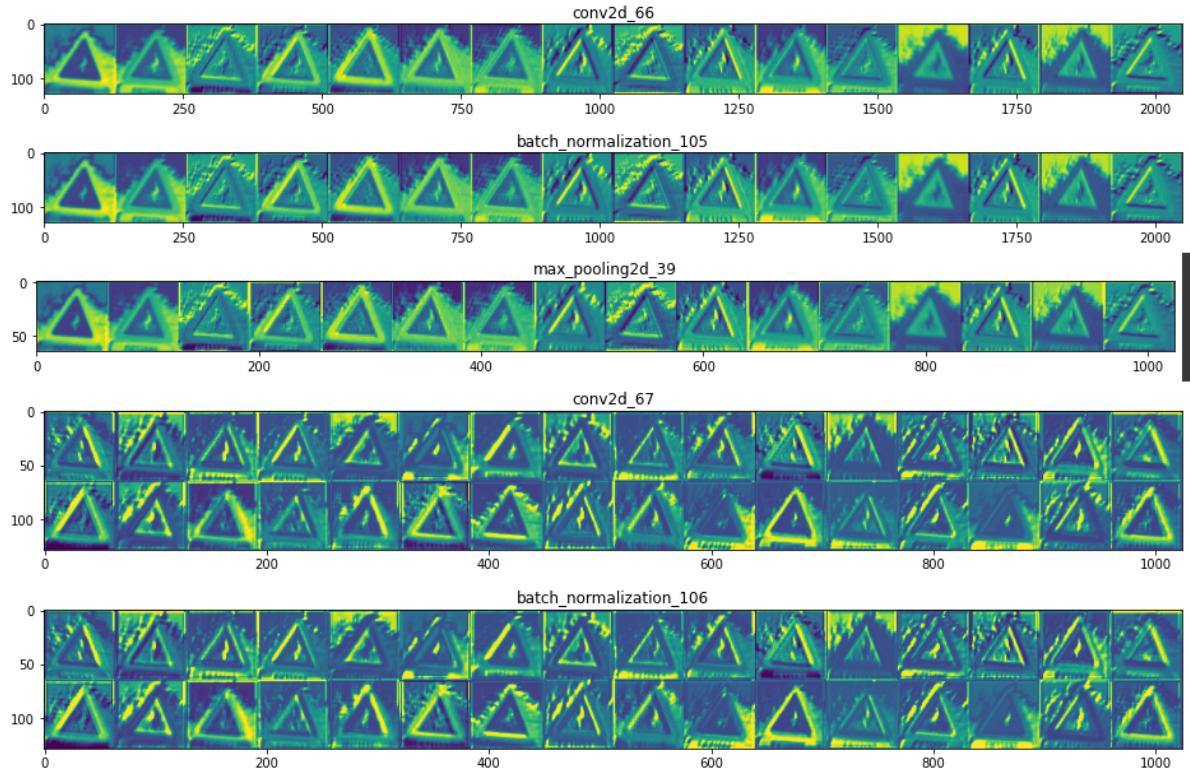
    # We will tile the activation channels in this matrix
    n_cols = n_features // images_per_row
    display_grid = np.zeros((size * n_cols, images_per_row * size))

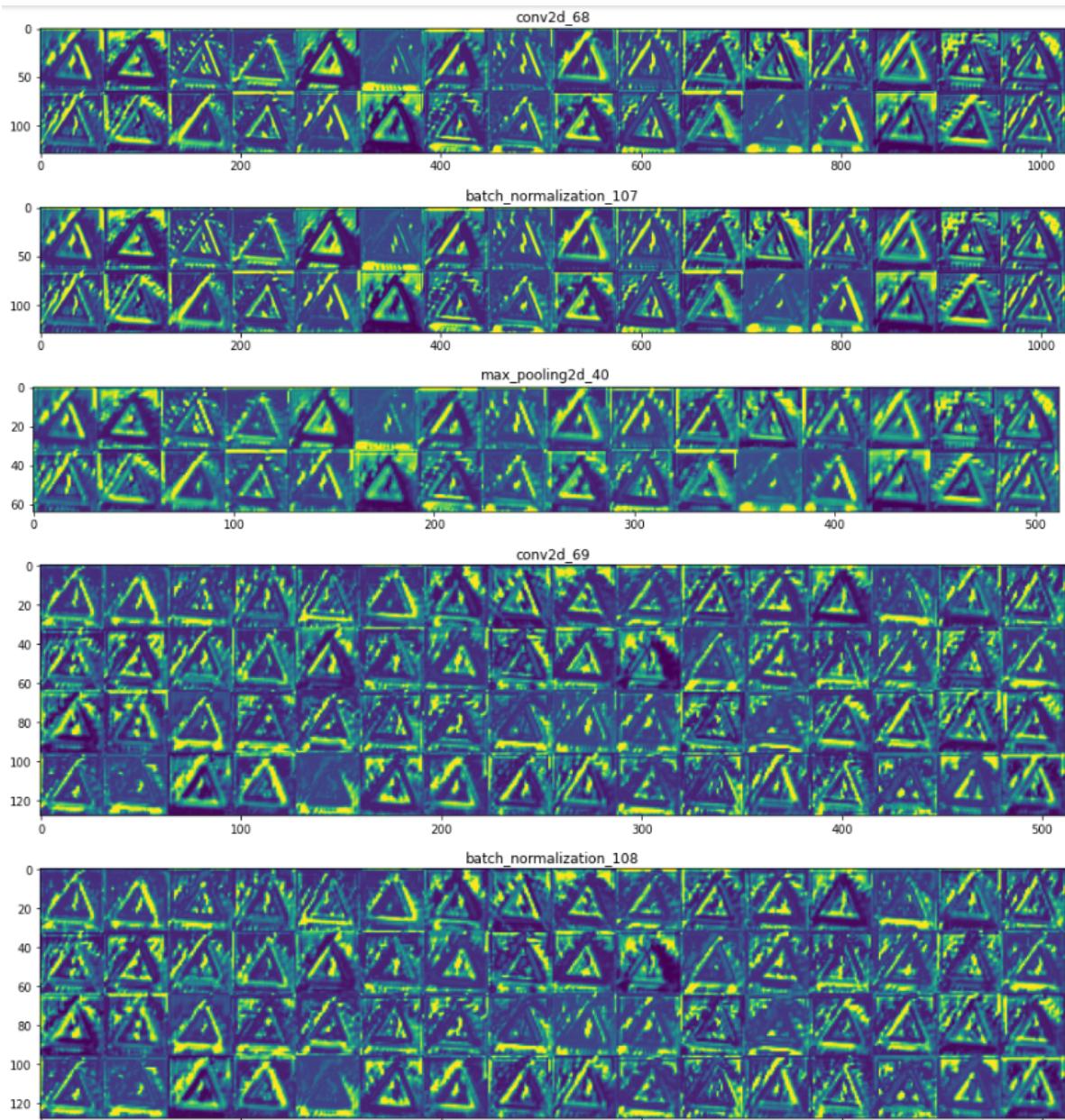
    # We'll tile each filter into this big horizontal grid
    for col in range(n_cols):
        for row in range(images_per_row):
            channel_image = layer_activation[0,
                                              :, :,
                                              col * images_per_row + row]
            # Post-process the feature to make it visually palatable
            channel_image -= channel_image.mean()
            channel_image /= channel_image.std()
            channel_image *= 64
            channel_image += 128
            channel_image = np.clip(channel_image, 0, 255).astype('uint8')
            display_grid[col * size : (col + 1) * size,
                         row * size : (row + 1) * size] = channel_image

    # Display the grid
    scale = 1. / size
    plt.figure(figsize=(scale * display_grid.shape[1],
                       scale * display_grid.shape[0]))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

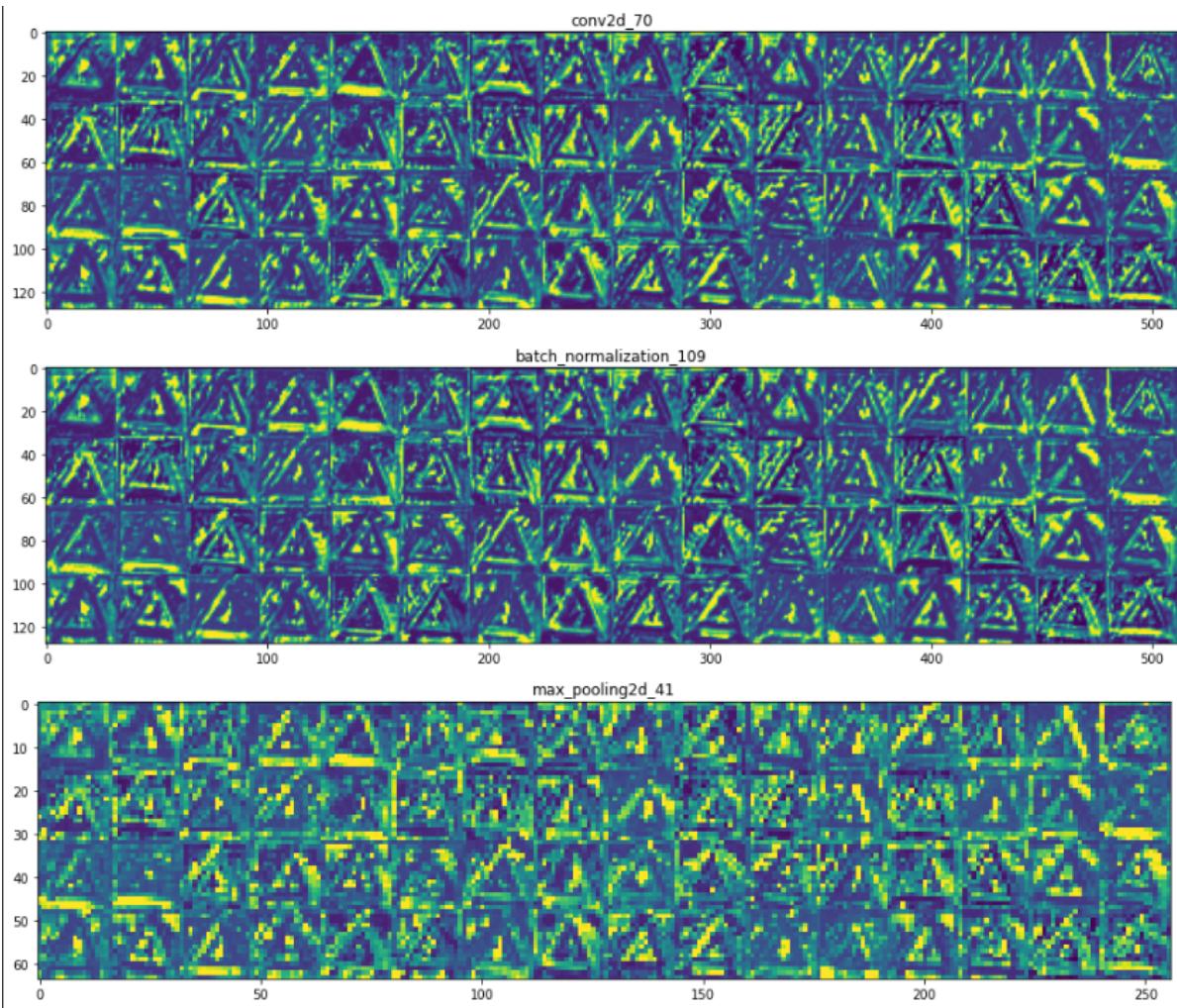
plt.show()

```





Αυτό που παρατηρούμε είναι το εξής. Ενώ αρχικά βλέπουμε ότι αναγνωρίζει low level χαρακτηριστικά(εντονο κιτρινο) ενώ καθώς βαθαίνει το δίκτυο φαίνεται να μαθαίνει όλο και πιο αφαιρετικά.

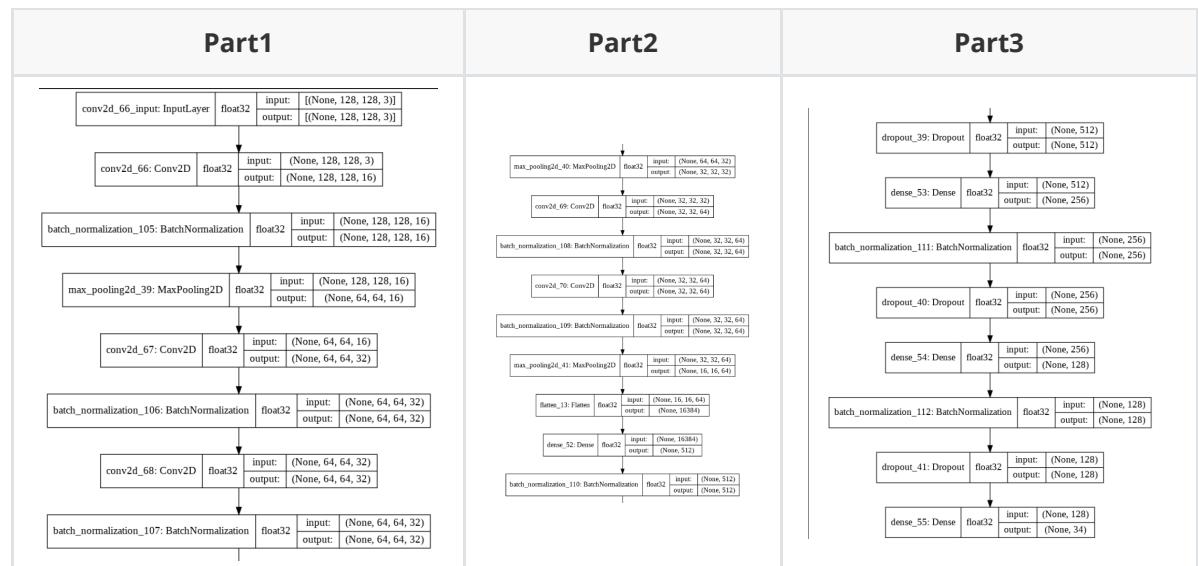


Τέλος, ο τρόπος για να πάρουμε σε μία εικόνα τις πληροφορίες για την αρχιτεκτονική μας είναι ο εξής:

```

drive
  -> imagedb
  -> imagedb_test
  -> sample_data
    -> model_plot.png
from keras.utils.vis_utils import plot_model
plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer_names=True, show_dtype=True)

```



## Πηγές-Αναφορές

---

1. Computer Vision: Algorithms and Applications 2nd Edition - Richard Szeliski - September 20, 2020, Springer
2. Διαφάνιες 10ου μαθήματος eclass: Deep Learning
3. NVIDIA Glossary
4. Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow - Concepts, Tools, and Techniques to Build Intelligent Systems - O'Reilly Media (2019)
5. Artificial Intelligence By Example-Acquire advanced AI, machine learning, and deep learning design skills-Denis Rothman 2020 Packt Publishing(Chapter 13:Visualizing Networks with TensorFlow 2.x and TensorBoard)
6. <https://github.com/digantamisra98/Mish>
7. <https://colab.research.google.com/github/fchollet/deep-learning-with-python-notebooks/blob/master/5.4-visualizing-what-convnets-learn.ipynb#scrollTo=FydxbyMFb6w>
8. <https://arxiv.org/pdf/1908.08681.pdf>
9. <https://arxiv.org/pdf/1710.05941v1.pdf>
10. <https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6>
11. <https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/>
12. <https://data-flair.training/blogs/keras-loss-functions/>
13. <https://arxiv.org/pdf/1804.07612.pdf>
14. <https://keras.io/>

Για πρόσβαση στα αποθηκευμένα μοντέλα χρησιμοποιήστε τα παρακάτω λινκ:

- Μη-προεκπαιδευμένο μοντέλο: <https://drive.google.com/file/d/15d2Pu2nbeO4cRSWhdQhcvhXwiRh9Wm0/view?usp=sharing>
- Προεπιδευμένο μοντέλο: <https://drive.google.com/file/d/10H4jWFWhzMwW96d0a36mdnNU0spSkMW2/view?usp=sharing>