Νευρωνικά Δίκτυα Εργασία 1

Παναγιώτης Καρβουνάρης

26 Νοεμβρίου 2023

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Στόχος της Εργασίας

Στόχος της εργασίας είναι το $classification$ της $cifar-10$ $dataset$. Η $cifar-10$ περιέχει 60.000 φωτογραφίες $32x32$ $pixels$, οι οποίες ομαδοποιούνται σε 10 διαφορετικές κλάσεις. Αρχικά, χρησιμοποιήθηκαν γνωστοί αλγόριθμοι $classification$ (k $nearest$ $neighbour$ $classifier$ και $centroid$ $classifier$) για την λύση του παραπάνω προβλήματος. Στην συνέχεια, δημιουργήθηκε ένα νευρωνικό δίκτυο MLP ($multi-layer$ $perceptron$) ($from$ $scratch$) για να επιλύσει το πρόβλημα. Σκοπό της παρακάτω ανάλυσης αποτελεί η σύγκριση των διαφορετικών τρόπων $classification$ ενός $dataset$ και η εξικοίωση με τον τρόπο λειτουργίας ενός νευρωνικού δικτύου.

1.2 Μαθηματικό Υπόβαθρο

Για την δημιουργία του νευρωνικού δικτύου χρησιμοποιήθηκε η μαθηματική ανάλυση από το βιβλίο $Haykin_NNETs_Part1$. Τόσο $forward$ όσο και το $back-propagation$ είναι υλοποιημένα με βάση την λογική των δ συναρτήσεων του βιβλίου. Για $antivation$ $function$ στα $hidden$ $layer$ χρησιμοποιήθηκε η $ReLU$ ενώ για το $output$ $layer$ χρησιμοποιήθηκε η $Softmax$. Ως $error$ $fuction$ χρησιμοποιήθηκε η $Categorical$ $Cross$ $Entropy$, η οποία έχει πολύ καλά αποτελέσματα σε προβλήματα $classification$ πολλών κλάσεων.

1.3 Διευκρηνίσεις Κώδικα

Η υλοποίηση τόσο του νευρωνικού δικτύου όσο και των αλγορίθμων *machine learning* έγινε σε *python*. Το *dataset* που χρησιμοποίησα είναι το $cifar-10$. Τα αποτελέσματα των προσομοιώσεων παραθέτονται παρακάτω μαζί με τις διάφορες αλλαγές και τα προβλήματα που υπήρχαν στην διάρκεια της ανάπτυξης του κώδικα. Κατά τα παραδείγματα που θα ακολουθήσουν, η αρχική αρχιτεκτονική που χρησιμοποιήθηκε ήταν, 10 νευρώνες στο *output layer* και 3072 νευρώνες στο πρώτο *hidden layer*. Στα τελευταία παραδείγματα έχει προσθεθεί ανάμεσά τους ένα επιπλέον *hidden layer* με 200 νευρώνες. Στην αρχή είχα αφοσιωθεί στον κώδικα και στις τιμές στο *terminal* και δεν έβγαζα *plot* γι' αυτό έβαλα τις τιμές στην αναφορά.

Σημείωση: Θα ήθελα να δοκιμάσω και με άλλον αριθμό στους νευρώνες του ενδιάμεσου *layer* (π.χ 256, 128, 64). Επίσης, στην αναφορά έχω βάλει μόνο μερικά απο τα παραδείγματα που έτρεξα και ολοκληρώθηκαν, επειδή μερικά τα σταματούσα γιατί έβρισκα μία βελτίωση ή έβγαζαν όχι καλά αποτελέσματα, οπότε τα σταματούσα και πήγαινα για *debug*.

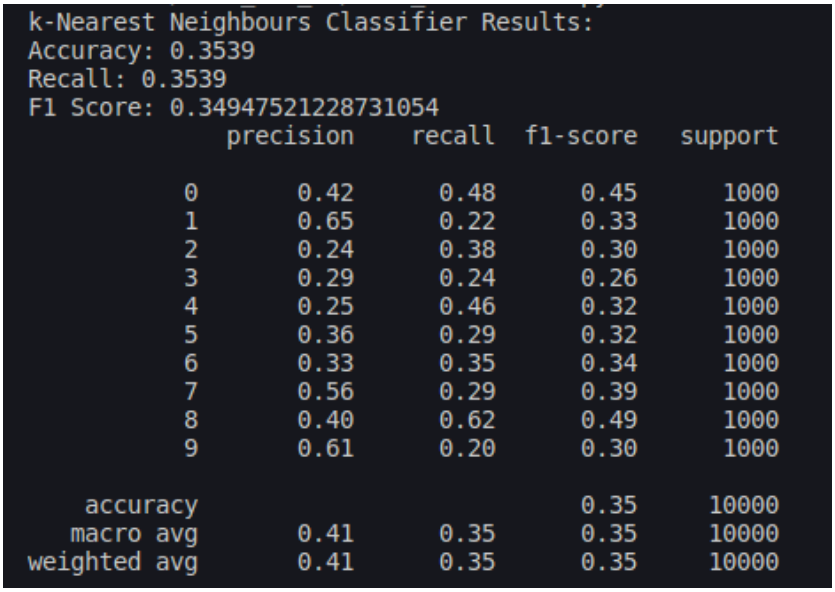
Κεφάλαιο 2

Αλγόριθμοι

2.1 $k - nearest\ neighbour$

Ο αλγόριθμος τοποθετεί όλο το *dataset* σε ένα $2D$ διάγραμμα με την χρήση του PCA και έπειτα όταν τοπεθετεί ένα καινούργιο στοιχείο στο διάγραμμα βλέπει ένας συγκεκριμένο αριθμό γειτόνων και η κλάση στην οποία ανήκουν οι περισσότεροι γείτονες, είναι και η κλασή του καινούργιου στοιχείου.

Ο αλγόριθμος με $k = 1$ βγάζει καλύτερα αποτελέσματα, καθώς εμφανίζει $accuracy = 0.35$ έναντι $accuracy = 0.33$ του αλγορίθμου που τρέχει με $k = 3$. Το *recall* εμφανίζει παρόμοια συμπεριφορά με το *accuracy* για τις δύο περιπτώσεις.Παρακάτω φαίνονται αναλυτικά τα αποτελέσματα του αλγορίθμου με $k = 1$ και $k = 3$.

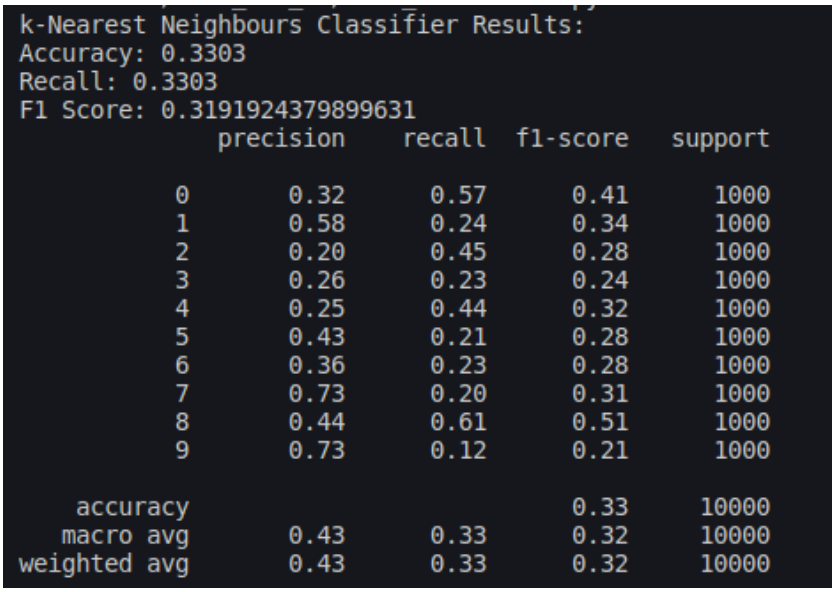


```
k-Nearest Neighbours Classifier Results:
Accuracy: 0.3539
Recall: 0.3539
F1 Score: 0.34947521228731054
              precision    recall  f1-score   support

           0       0.42      0.48      0.45      1000
           1       0.65      0.22      0.33      1000
           2       0.24      0.38      0.30      1000
           3       0.29      0.24      0.26      1000
           4       0.25      0.46      0.32      1000
           5       0.36      0.29      0.32      1000
           6       0.33      0.35      0.34      1000
           7       0.56      0.29      0.39      1000
           8       0.40      0.62      0.49      1000
           9       0.61      0.20      0.30      1000

    accuracy                           0.35     10000
   macro avg       0.41      0.35      0.35     10000
weighted avg       0.41      0.35      0.35     10000
```

Σχήμα 2.1: $k - nn\ classifier,\ k = 1$



```
k-Nearest Neighbours Classifier Results:
Accuracy: 0.3303
Recall: 0.3303
F1 Score: 0.3191924379899631
              precision    recall  f1-score   support

           0       0.32      0.57      0.41      1000
           1       0.58      0.24      0.34      1000
           2       0.20      0.45      0.28      1000
           3       0.26      0.23      0.24      1000
           4       0.25      0.44      0.32      1000
           5       0.43      0.21      0.28      1000
           6       0.36      0.23      0.28      1000
           7       0.73      0.20      0.31      1000
           8       0.44      0.61      0.51      1000
           9       0.73      0.12      0.21      1000

    accuracy                           0.33     10000
   macro avg       0.43      0.33      0.32     10000
weighted avg       0.43      0.33      0.32     10000
```

Σχήμα 2.2: $k - nn\ classifier,\ k = 3$

2.2 *Nearest Centroid*

Ο αλγόριθμος τοποθετεί όλο το *dataset* σε ένα $2D$ διάγραμμα με την χρήση του PCA και βρίσκει το κέντρο της κάθε κλάσεις πάνω στο διάγραμμα με βάση την τοποθεσία των ήδη γνωστών στοιχείων της κλάσης. Όταν τοποθετείται ένα καινούργιο στοιχείο στο διάγραμμα, απλά υπολογίζεται η ευκλείδια απόσταση μεταξύ των συντεταγμένων του καινούργιου στοιείου και των κεντρών των κλάσεων, όποιο κέντρο είναι πιο κοντά αυτή είναι και η κλάση του καινούργιου στοιχείου.

Παρατηρούμε ότι αυτός ο αλγόριθμος στην συγκεκριμένη περίπτωση αποδίδει χειρότερα και από τις δύο περιπτώσεις του $k-nn$ αλγορίθμου. Η ακρίβειά του είναι $accuracy = 0.28$. Παρακάτω φαίνονται αναλυτικά τα αποτελέσματα του αλγορίθμου.

```
Nearest Centroid Classifier Results:
Accuracy: 0.2774
Recall: 0.2774
F1 Score: 0.25408598962127454
              precision    recall  f1-score   support

           0       0.27      0.54      0.36      1000
           1       0.28      0.19      0.22      1000
           2       0.28      0.11      0.16      1000
           3       0.27      0.06      0.09      1000
           4       0.28      0.12      0.17      1000
           5       0.27      0.29      0.28      1000
           6       0.22      0.54      0.31      1000
           7       0.27      0.17      0.20      1000
           8       0.42      0.37      0.39      1000
           9       0.33      0.41      0.36      1000

    accuracy                           0.28     10000
   macro avg       0.29      0.28      0.25     10000
weighted avg       0.29      0.28      0.25     10000
```

Σχήμα 2.3: *nearest centroid*

Κεφάλαιο 3

Νευρωνικό Δίκτυο

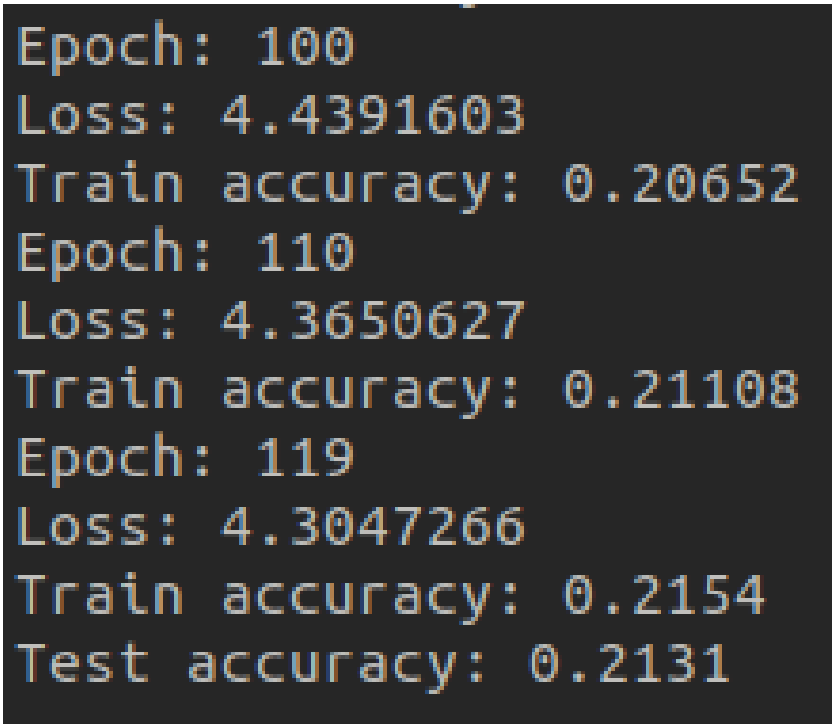
3.1 Δημιουργία Κώδικα

Ο κώδικας για το νευρωνικό δίκτυο υλοποιήθηκε σε *python* με την χρήση του $VSCode$ ως IDE. Αρχικά, έπρεπε να υλοποιηθούν οι βασικές συναρτήσεις και κλάσεις για την δημιουργία ενός ποιοτικού κώδικα, ώστε να είναι ευκολότερο να τρέχεις διάφορα παραδείγματα στην συνέχεια. Κατα την διάρκεια της συγγραφής του κώδικα μερικά προβλήματα σχετικά με την σωστή υλοποίηση των μαθηματικών μέσα στον κώδικα. Τα αρχικά αποτελέσματα του κώδικα όταν ήταν έτοιμος να τρέξει χωρίς *error* και χωρίς κάποιο μεγάλο βασικό λάθος στην υλοποίηση ήταν λίγο μπερδεμένα. Καθώς προχωρούσε ο κώδικας, βρήκα πολλές βελτιώσεις, μερικά μικρά λάθη στην υλοποίηση του κώδικα και μερικούς τρόπους που βελτίωσαν το *accuracy* του νευρωνικού.

3.2 Πρώτες Προσπάθειες

1ο παράδειγμα

$learning\ rate = 1e-4$, $epoch = 120$, $mini\ batch = 200$

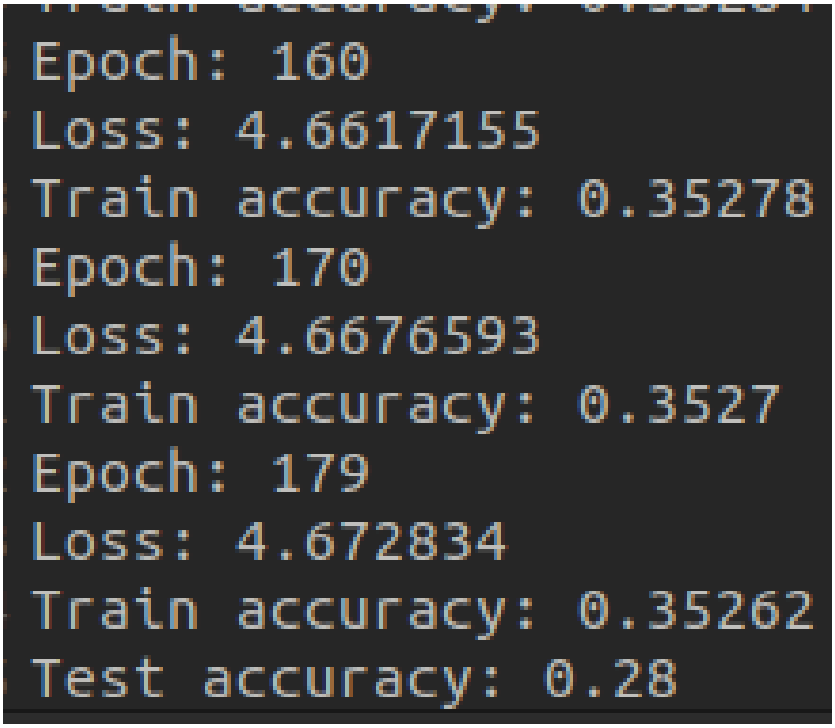


Σχήμα 3.1: $learning\ rate = 1e-4$, $epoch = 120$, $mini\ batch = 200$

Επειδή όπως φάινεται και παραπάνω φαίνεται να συγκλίνει αλλά με τόσο χαμηλό *learning rate* και μόνο 120 *epoch* οπότε στο επόμενο παράδειγμα, αύξησα τον αριθμό των *epoch* και το *learning rate* για να δούμε αν υπάρχει σύγκλιση με καλύτερο αποτέλεσμα.

2ο παράδειγμα

$learning\ rate = 1e-2$, $epoch = 0-50$, $learning\ rate = 1e-3$, $epoch = 50-100$, $learning\ rate = 1e-4$, $epoch = 100-200$, $mini\ batch = 200$



Σχήμα 3.2: $learning\ rate = 1e{-}2 -> epoch = 0{-}50$, $learning\ rate = 1e{-}3 -> epoch = 50 - 100$, $learning\ rate = 1e - 4 -> epoch = 100 - 200$, $mini\ batch = 200$

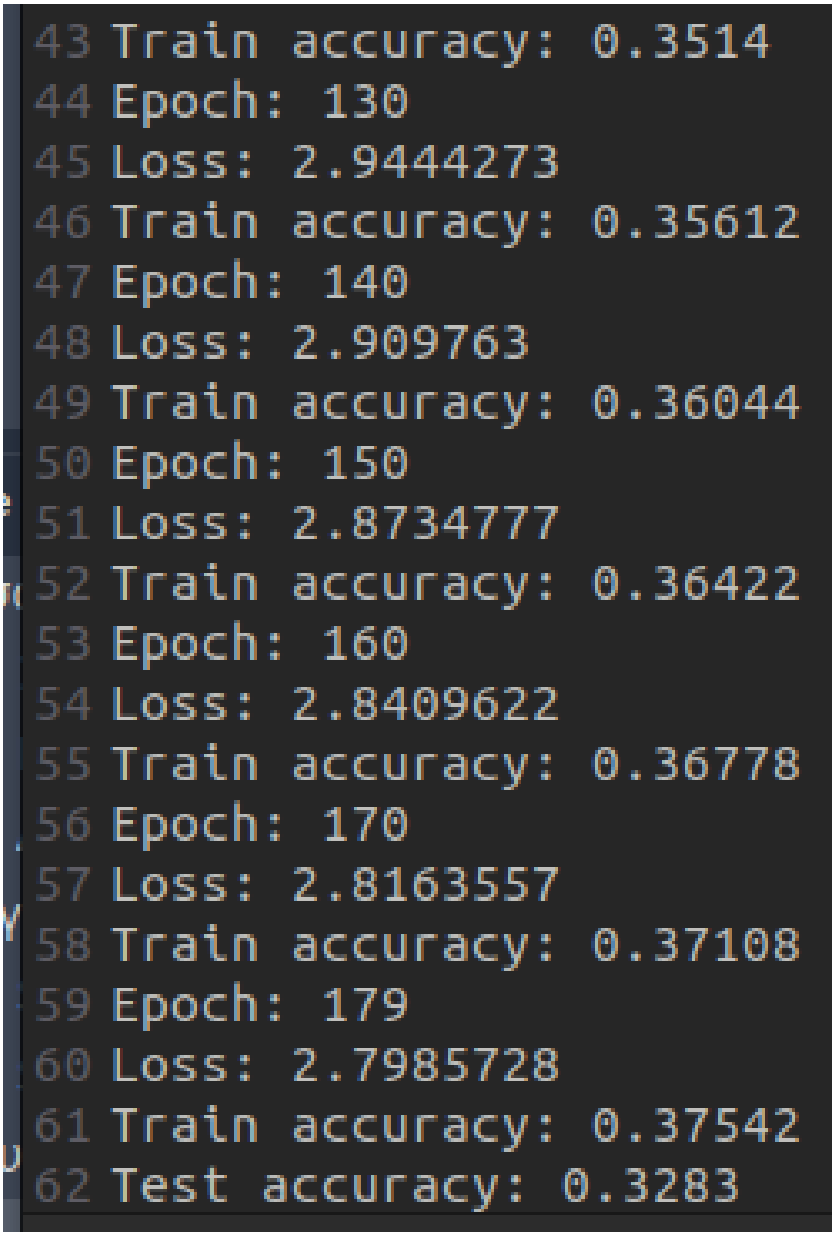
Στην παραπάνω περίπτωση υπήρχαν πολλά ανεβοκατεβάσμα στην απόδοση του νευρωνικού, γεγονός που δεν μπορούσε να εξηγηθεί απλά από κάποια παράμετρο, δηλαδή κάποιο πιθανό $overfiting$.

3.3 Βελτιώσεις και καινούργιες προσπάθειες

Στα παραπάνω αποτελέσματα η αστάθεια είναι ιδιαίτερα έντονη, γεγονός που δύσκολα συμβαίνει λόγω των παραμέτρων. Έπειτα από μερικές αλλαγές στον κώδικα (ανάστροφοι πίνακες, αλλαγή στην σειρά μερικών πράξεων και κάποιες κανονικοποιήσεις αποτελεσμάτων) τα καινούργια αποτελέσματα είναι τα παρακάτω.

3ο παράδειγμα

$learning\ rate = 1e - 3$, $epoch = 180$ $mini\ batch = 200$



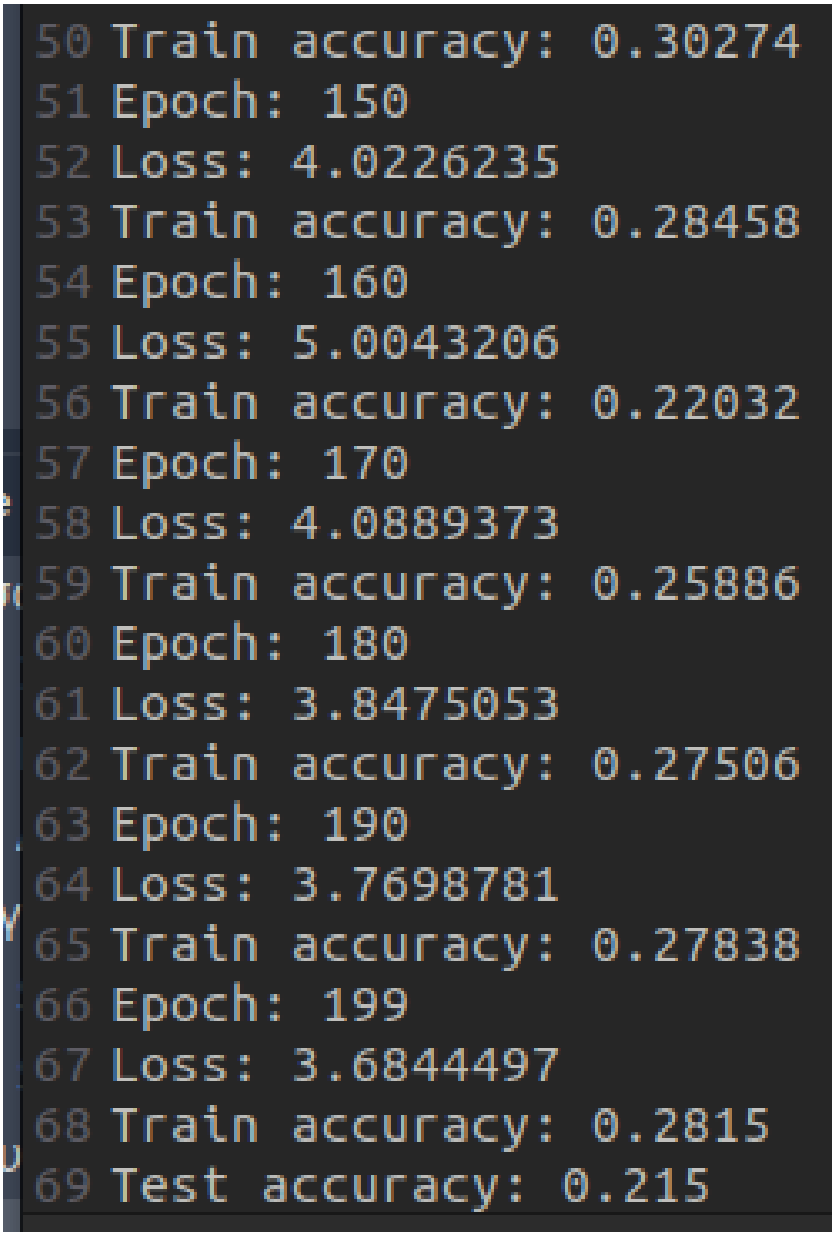
Σχήμα 3.3: $learning\ rate = 1e-3$, $epoch = 180$, $mini\ batch = 200$

Συνεχίζει να φαίνεται η αδυναμία του νευρωνικού. Σημαντική διαφοροποίηση εδώ είναι το γεγονός ότι δεν υπάρχουν πλέον σκαμπανεβάσματα της απόδοσης σε τυχαία *epoch*, που σημαίνει ότι αλλαγές στον κώδικα τον βελτίωσαν και στα

προηγούμενα παραδείγματα δεν λειτουργούσε σωστά. Επομένως στην επόμενη προσπάθεια άλλαξα μερικές παραμέτρους για να βελτιώσω την απόδοση.

4ο παράδειγμα

$learning\ rate = 1e-2 -> epoch = 0-10$ $learning\ rate = 1e-3 -> epoch = 10-100$ $learning\ rate = 1e-4 -> epoch = 100-200$ $batch = 200$



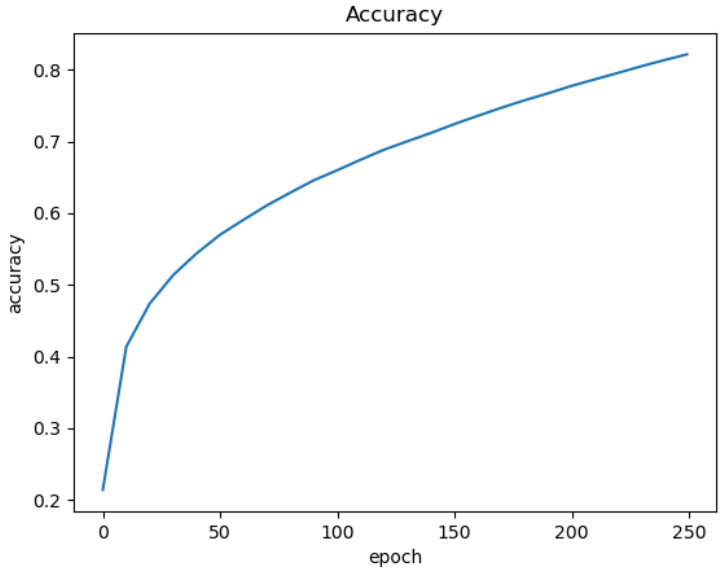
Σχήμα 3.4: $learning\ rate = 1e-2 -> epoch = 0-10$ $learning\ rate = 1e-3 -> epoch = 10-100$ $learning\ rate = 1e-4 -> epoch = 100-200$ $batch = 200$

3.4 Εύρεση λαθών και τελικές προσπάθειες

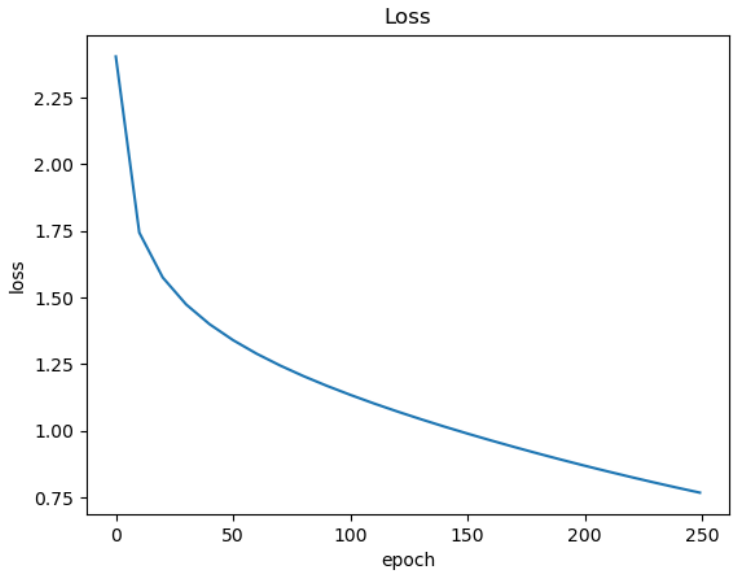
Στο τελευταίο παράδειγμα, εμφανίστηκε η αστάθεια στις τιμές κατά την διάρκεια του *training*, γεγονός που δημιουργήθηκε μάλλον από τις αλλαγές στις τιμές του *learning rate*. Όμως, καθώς έκανα δύο τρεις αποτυχημένες προσπάθειες να βελτιώσω την κατάσταση, παρατήρησα ένα σημαντικό λάθος στην υλοποιήση του *backpropagation*, ένα εσωτερικό γινόμενο γινόταν με λάθος τρόπο και αυτό επηρέαζε έντονα το αποτέλεσμα, αλλά το λάθος δεν βαρούσε *compile error* επειδή το λάθος δεν φαινόταν όταν είχα μόνο ένα *hidden*. Επομένως έπειτα από αρκετές διορθώσεις τα αποτελέσματα, άρχισαν να βελτιώνονται πολύ.

5ο παράδειγμα

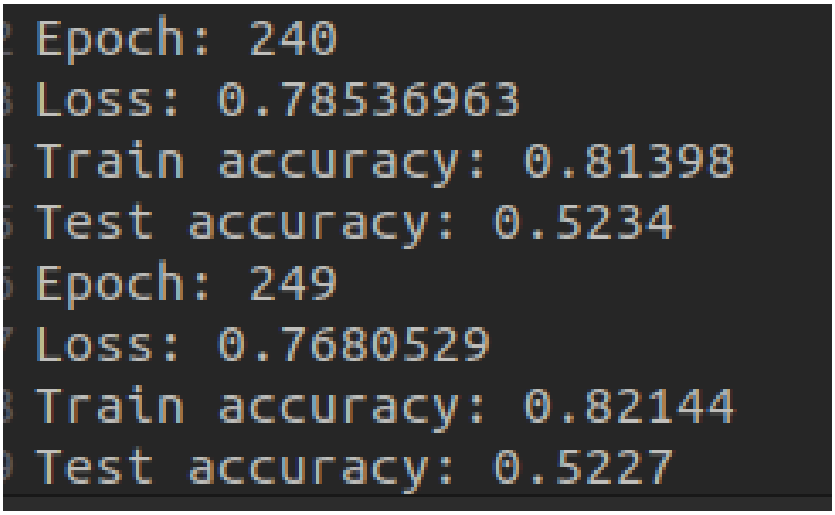
$learning\ rate = 1e-2$, $epoch = 250$ $batch = 400$



Σχήμα 3.5: *Accuracy for learning rate* $= 1e-2$, $epoch = 250$ $batch = 400$



Σχήμα 3.6: *Loss for learning rate* $= 1e-2$, *epoch* $= 250$ *batch* $= 400$



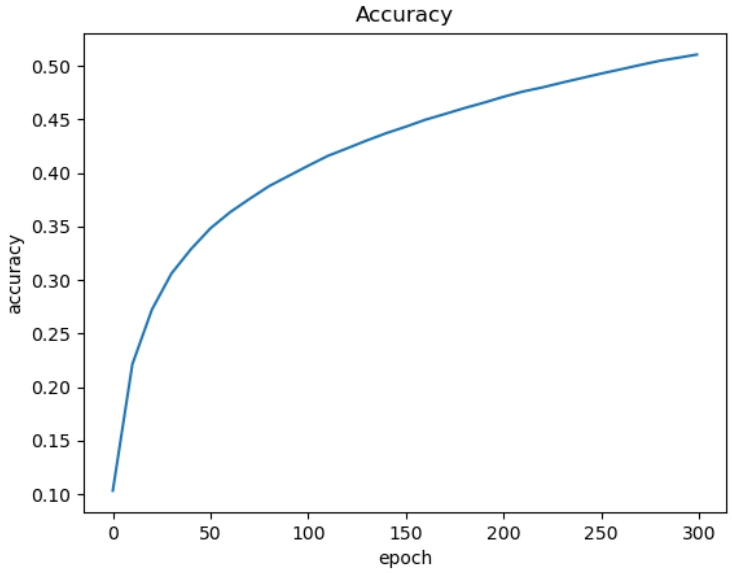
Σχήμα 3.7: *learning rate* $= 1e-2$, *epoch* $= 250$ *batch* $= 400$

Κάπου εδώ, το νευρωνικό φαίνεται να δουλεύει καλά χωρίς προβλήματα λογικής και μαθηματικών. Τα αποτελέσματα αρκικά είναι ικανοποιητικά, όμως

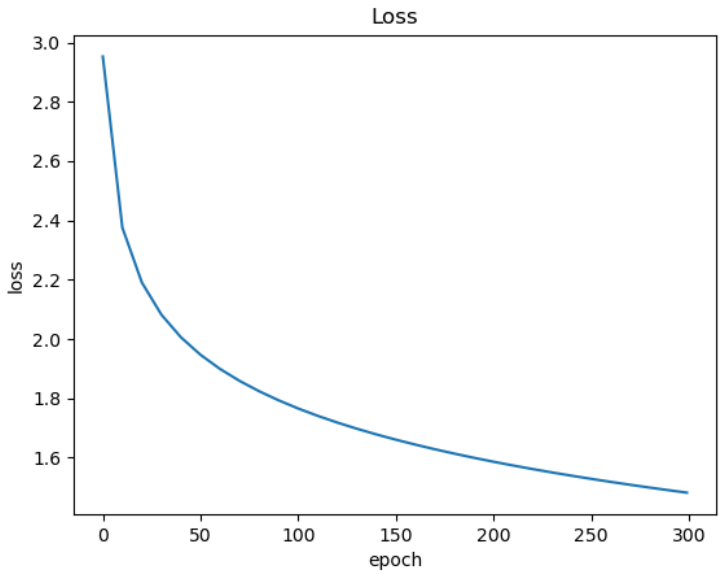
είναι ξεκάθαρο ότι μετά από ένα σημείο παθαίνει $overfiting$. Οπότε καινούργιο παράδειγμα για να αντιμετωπίσουμε το $overfiting$.

6ο παράδειγμα

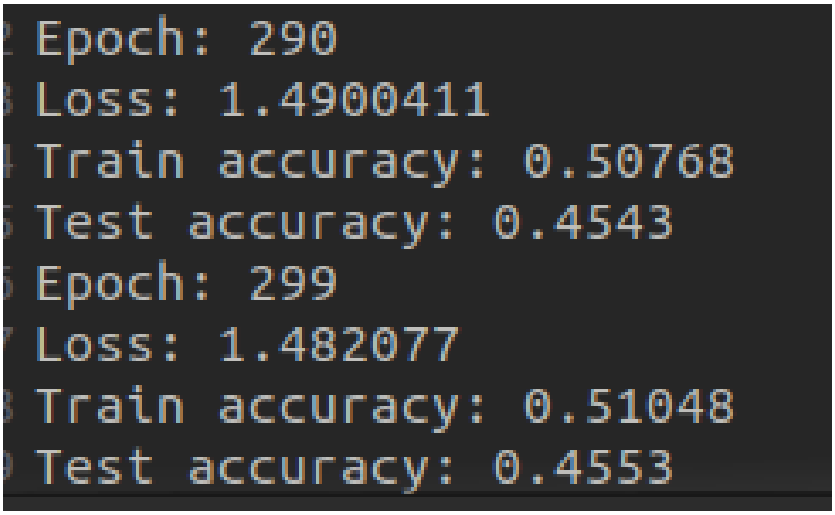
$learning\ rate = 1e-3$, $epoch = 300$ $batch = 400$



Σχήμα 3.8: $Accuracy\ for\ learning\ rate = 1e-3$, $epoch = 300$ $batch = 400$



Σχήμα 3.9: *Loss for learning rate* $= 1e-3$, *epoch* $= 300$ *batch* $= 400$



Σχήμα 3.10: *learning rate* $= 1e-3$, *epoch* $= 300$ *batch* $= 400$

Σε αυτό το παράδειγμα ίσως να χρειαζόταν περισσότερα *epoch* όμως το $overfiting$ είχε αρχίσει ήδη να εμφανίζεται στο *epoch* $= 300$.

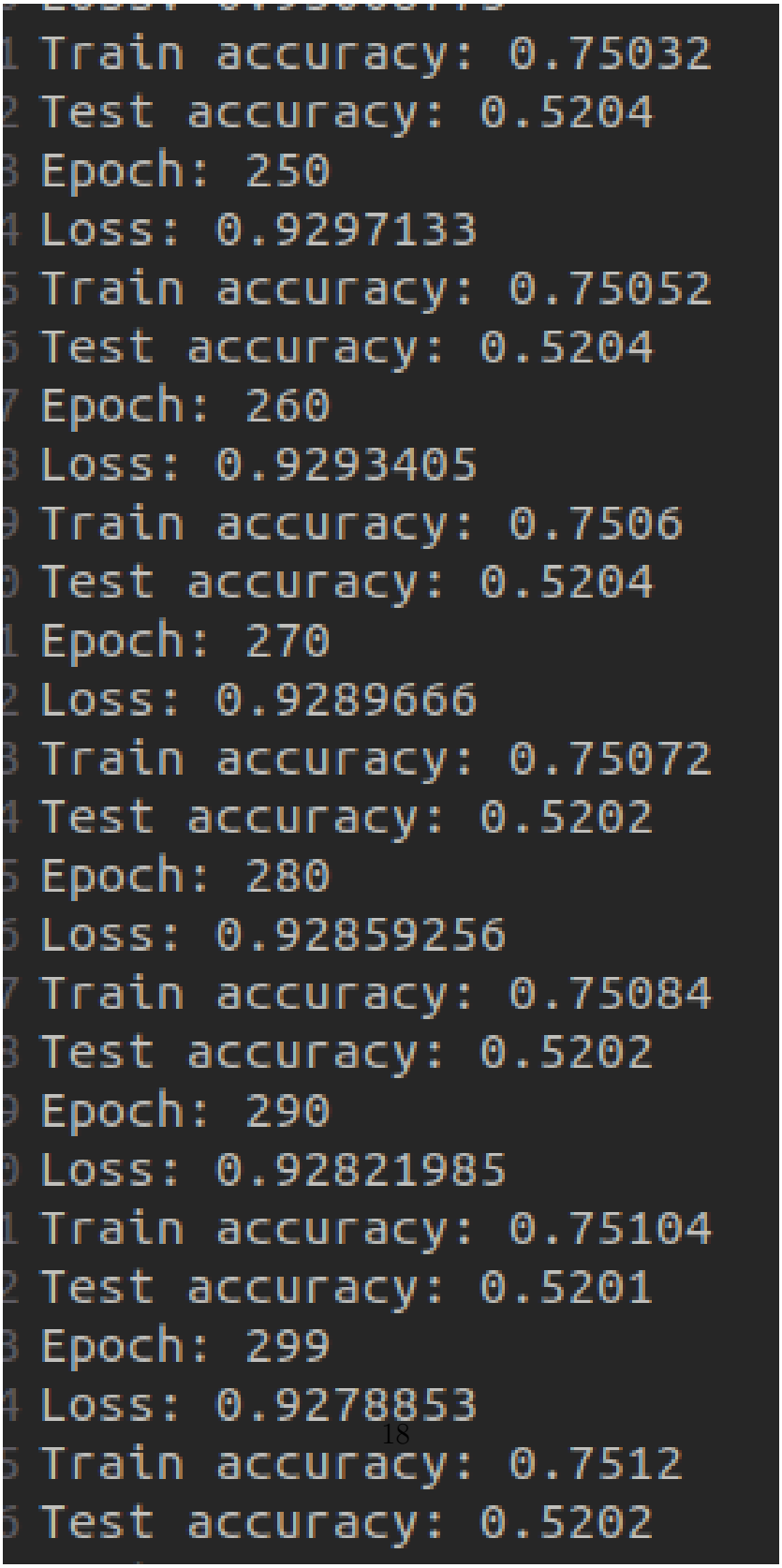
3.5 Τελική προσπάθεια

Μετά από όλες τις προσπάθειες και μία μικρή προσαρμογή στον κώδικα (στα προηγούμενα παραδείγματα ανανέωνα τα βάρη των $hidden\ layer$ με βάση τα ήδη ανανεωμένα βάρη του $output\ layer$, μετά από μία άσκηση που είχαμε κάνει στην τάξη παρατήρησα ότι τα βάρη του $hidden\ layer$ ανανεώνονται με βάση τα παλιά βάρη του $output\ layer$). Η αλλαγή αυτή φάνηκε να έχει καλύτερα αποτελέσματα όσον αφορά το $overfiting$, καθώς δεν εφμανίζεται πλέον κατά την διάρκει του $training$, αλλά μετά από κάποιο σημείο το φαινόμενο γίνεται και πάλι έντονο.

Γενικά στην διάρκεια ανάπτυξης του κώδικα έβρισκες αρκετούς τρόπους οι οποίοι θα μπορούσαν να βοηθήσουν στην αντιμετόπιση προβλημάτων όπως το $overfiting$.Το τελευταίο παράδειγμα είναι παρακάτω (υπήρξε ένα λάθος με το $plot$ των δύο διαγραμμάτων και δυστυχώς δεν είχα τον χρόνο να το ξανατρέξω και να πάρω τα σωστά $plot$ για να τα βάλω στην αναφορά)

7ο παράδειγμα

$learning\ rate = 1e-2 -> epoch = 0-100$ $learning\ rate = 1e-3 -> epoch = 100-200$ $learning\ rate = 1e-4 -> epoch = 200-300$ $batch = 250$



Σχήμα 3.11: $learning\ rate = 1e-2 -> epoch = 0-100$ $learning\ rate = 1e-3 -> epoch = 100-200$ $learning\ rate = 1e-4 -> epoch = 200-300$

3.6 Συμπεράσματα

Κατά την ανάπτυξη του κώδικα για το νευρωνικό δίκτυο έπρεπε να μπορείς να γράφεις καθαρό και εύκολα διαχειρήσιμο κώδικα και παράλληλα να κρατάς την μαθηματική αυστηρότητα στις πράξεις, επειδή μια πολύ μικρή λεπτομέρεια όπως ένα ανάστροφο μπορούσε να σου χαλάσει το αποτέλεσμα. Τα νευρωνικά τέτοιου είδους όπως φαίνεται και από τα αποτελέσματα, έχουν πολύ καλύτερη επίδοση από τους αλγορίθμους *machine learning* που παρουσιάστηκαν παραπάνω.

Βήματα για την βελτίωση των αποτελεσμάτων:

1) Στα τελευταία *epoch* ακόμα μικρότερα *learning rate* (π.χ $1e-5$)

2) Χρήση μικρότερου *batch* (π.χ 100)

3) Χρήση της μεθόδου $L2$ κανονικοποίησης, που είναι ένας τρόπος περιορισμού της εμφάισης του $overfiting$. Πρακτικά τα βάρη ανανεώνονται σύμφωνω με αυτόν τον κανόνα $w = w - lr(dw - a * w)$, όπου a παίρνει μικρές τιμές $a = 1e-3$.

4) Δοκιμή διαφορετικών αρχιτεκτονικών. Στο ενδιάμεσο *layer* άλλος αριθμός νευρώνων (π.χ 64, 128, 512)

Δεν μπορούσα να τρέξω πολλά παραδείγματα, επειδή εξαιτίας της δικιάς μου υλοποίησης του κώδικα, χρειάστηκαν πολλά παραδείγματα για *debug*, επομένως όταν κατέληξα να έχω έναν λειτουργικό κώδικα δεν είχα πολύ χρόνο να δοκιμάσω πολλά. Για 300 *epoch* χρειαζόταν περίπου 2,5-3 ώρες. Πολύ ενδιαφέρουσα εργασία και πολύ ικανοποιητικό να βλέπεις το νευρωνικό που έφτιαξες από το μηδέν να δουλεύει και να εκπαιδεύεται.