Neural Networks
&
Deep Learning

3Η ΥΠΟΧΡΕΩΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Μυλωνάς | 10027 | 09.01.2023

Word Count: 2693

Πίνακας περιεχομένων

Εισαγωγή 3
Τεχνικές Προδιαγραφές 4
Υλοποίηση 5
KNN, NCC 5
KNN Cross Validation 5
Άλλες τεχνικές βελτιστοποίησης της ακρίβειας 5
RBF Neural Network 6
Autoencoder 6
Γραφική αναπαράσταση μετρήσεων 7
Σχολιασμός Αποτελεσμάτων 13
KNN vs NCC Metrics 13
CROSS VALIDATION VS KNN Metrics 13
RBFNN Metrics 14
Autoencoder Metrics 14
KNN, NCC, KNN CV, RBFNN vs Autoencoder Metrics 14
Παράρτημα 15
(A) Πίνακες Μετρήσεων 15
(B) Κώδικας 20
KNN & NCC 20
KNN Cross Validation 21
RBFNN 22
AUTOENCODER 24
Βιβλιογραφία 25

Εισαγωγή

Στην συγκεκριμένη εργασία ζητήθηκαν η υλοποίηση ενός προγράμματος σε οποιαδήποτε γλώσσα προγραμματισμού, το οποίο να συγκρίνει την απόδοση του κατηγοριοποιητή πλησιέστερου γείτονα με 1 και 3 γείτονες (K-Nearest Neighbor Classifier ή KNN) με αυτή του κατηγοριοποιητή πλησιέστερου κέντρου (Nearest Class Centroid Classifier ή NCC) και η υλοποίηση ενός Radial Basis Function Neural Network (RBFNN) και η σύγκριση της απόδοσης αυτού με τους παραπάνω κατηγοριοποιητές. Το RBFNN εκπαιδεύεται για να διαχωρίζει χειρόγραφα ψηφία. Το πρόβλημα αυτό εμπίπτει στη κατηγορία της επίλυσης προβλήματος κατηγοριοποίησης πολλών κλάσεων.

Ο κατηγοριοποιητής KNN υπολογίζει τα πλησιέστερα K σημεία και κατηγοριοποιεί το στοιχείο που μελετάμε με βάση τη συχνότητα εμφάνισης των κατηγοριών αυτών των K σημείων. Είναι σαφές πως για να μην υπάρχει ισοψηφία, επιλέγουμε την παράμετρο K να είναι περιττός αριθμός.

Ο κατηγοριοποιητής NCC αρχικά διαχωρίζει σε κλάσεις τα πιθανά αποτελέσματα (10 στη συγκεκριμένη περίπτωση). Στην συνέχεια υπολογίζει το κέντρο (centroid) της κάθε κλάσης με τη χρήση του μέσου όρου. Τελικά, συγκρίνει την απόσταση του σημείου που μελετάμε από αυτά τα κέντρα, και το κατηγοριοποιεί το σημείο στην κλάση που η απόσταση αυτή είναι η ελάχιστη.

Το RBFNN είναι αρκετά όμοιο τόσο με τον MLP όσο και με το SVM με kernel = RBF. Πρόκειται για ένα νευρωνικό δίκτυο feed forward, με 3 τύπους από στρώσεις (layers): εισόδου (input), κρυφούς (hidden) και εξόδου (output). Είναι πλήρως συνδεδεμένο νευρωνικό δίκτυο και χρησιμοποιείται τόσο για προβλήματα κατηγοριοποίησης (classification) όσο και για προβλήματα παλινδρόμησης (regression). Ουσιαστικά το δίκτυο αυτό έχει ακριβώς 3 στρώσεις νευρώνων, σε αντίθεση με το MLP μοντέλο που μπορεί να έχει παραπάνω. Η κρυφή στρώση χρησιμοποιεί την RBF συνάρτηση για να υπολογίσει τα βάρη των νευρώνων. Χρησιμοποιούμε την Γκαουσσιανή Συνάρτηση στο κρυφό επίπεδο. Ο χώρος δειγμάτων διαιρείται σε σφαιρικές περιοχές (αλλά όχι αυστηρά σε κύκλους), οι οποίες μεταβάλλονται ανάλογα με τις υπερπαραμέτρους που εφαρμόζουμε. Αυτές είναι οι περιοχές επιρροής στον n-διάστατο χώρο. Το δίκτυο εκπαιδεύεται με διάφορους αλγορίθμους. Στο τελικό στάδιο του output layer γίνεται ο διαχωρισμός μέσω της συνάρτησης softmax. Η διαφορά του με το SVM μοντέλο με kernel = RBF είναι ότι έχει παραπάνω υπερπαραμέτρους που μπορούμε να τροποποιήσουμε.

Τέλος, έγινε απόπειρα υλοποίησης ενός αυτοκωδικοποιητή (autoencoder), για την επίλυση του προβλήματος κατηγοριοποίησης ψηφίων σε κλάσεις μέσω ανακατασκευής. Ο αυτοκωδικοποιητής προσπαθεί να κάνει compress την είσοδο αρχικά, και στη συνέχεια μέσω αυτής να ανακατασκευάσει την έξοδο όσο πιο κοντά στην είσοδο γίνεται. Αποτελείται από 2 κομμάτια, τον κωδικοποιητή (encoder) και τον αποκωδικοποιητή (decoder). Το πρώτο μετατρέπει το input σε μια κρυφή αναπαράσταση (latent representation) μέσω συμπίεσης (compression), και το δεύτερο παίρνει τη συμπιεσμένη μορφή και προσπαθεί να αναπαράγει την έξοδο.

Τεχνικές Προδιαγραφές

Για την υλοποίηση χρησιμοποιήθηκε Python ενώ ως περιβάλλον εργασίας το PyCharm.

Ως βάση δεδομένων επιλέχθηκε η MNIST[1] dataset, η οποία περιλαμβάνεται στο Keras[2]. Η MNIST αποτελεί μια συλλογή από χειρόγραφα ψηφία, τα οποία χρησιμοποιούνται για την εκπαίδευση αλγορίθμων επεξεργασίας εικόνας. Το Keras είναι ένα API βαθιάς μάθησης γραμμένο σε Python, που τρέχει πάνω στην πλατφόρμα μηχανικής μάθησης TensorFlow.

Για τους κατηγοριοποιητές (KNN, NCC, SVC) χρησιμοποιήθηκε η βιβλιοθήκη μηχανικής μάθησης scikit-learn ή sklearn[3].

Σαφώς χρησιμοποιήθηκαν και Python modules όπως NumPy και Time.

Η προδιαγραφές της συσκευής που χρησιμοποιήθηκε για την εκτέλεση του κώδικα και τη λήψη μετρήσεων είναι οι εξής:

- Processor: 12th Gen Intel(R) Core(TM) i7-1260P @ 2.10 GHz
- Ram memory: 32 GB
- System type: 64-bit operating system, x64-based processor

[1] MNIST Dataset: http://yann.lecun.com/exdb/mnist/
[2] Keras: https://keras.io/about/
[3] Sklearn: https://scikit-learn.org/stable/index.html

Υλοποίηση

KNN, NCC

Αρχικά εισάγουμε τις απαραίτητες βιβλιοθήκες, ώστε να μπορούμε να χρησιμοποιήσουμε τη βάση δεδομένων, διάφορες μεθόδους και τους κατηγοριοποιητές. Εναλλάσσοντας παραμέτρους στον αλγόριθμο, όπως τον τύπο της απόστασης σημείων ή τον αριθμό των γειτόνων στον KNN, παίρνουμε μετρήσεις σχετικά με το ποσοστό επιτυχίας και το χρόνο εκπαίδευσης και πρόβλεψης. Στη συγκεκριμένη εργασία λήφθηκαν μετρήσεις από 1 έως 25 γείτονες στον KNN και Ευκλείδεια και Manhattan απόσταση στον NCC (για σκοπούς απλής παρατήρησης). Τελικά, συγκρίνουμε τόσο τον χρόνο όσο και την ακρίβεια κάθε αλγορίθμου και καταλήγουμε σε συμπεράσματα.

KNN CROSS VALIDATION

Η βάση δεδομένων περιλαμβάνει 2 σετ, 1 για εκπαίδευση (train set) και 1 για testing (test set). Το train set περιλαμβάνει 50,000 στοιχεία, ενώ το test set περιλαμβάνει 10,000 στοιχεία. Μελετώντας τον KNN περαιτέρω, βλέπουμε πως μπορεί να εμφανιστεί το φαινόμενο overfitting, δηλαδή ο αλγόριθμος να μαθαίνει πολύ εύκολα το dataset που του δίνουμε και να προσαρμόζεται πλήρως σε αυτό, χωρίς να μπορεί να διαχειριστεί διαφορετικά datasets. Επομένως, χρησιμοποιήθηκε και η τεχνική k-fold Cross Validation (CV) για πιο ακριβή αποτελέσματα. Στην CV, ενώνουμε τα train & test sets σε ένα set 60,000 εικόνων. Στη συνέχεια χωρίζουμε σε k folds το συνολικό dataset και χρησιμοποιούμε 1 fold ως test set και τα υπόλοιπα k-1 folds ως train sets. Με αυτόν τον τρόπο εξασφαλίζουμε πως κάθε στοιχείο του συνολικού set των 60,000 στοιχείων θα έχει χρησιμοποιηθεί και για εκπαίδευση και για testing και συνεπώς πως ο αλγόριθμός μας μπορεί να διαχειρίζεται και διαφορετικά datasets.

Στη συγκεκριμένη εργασία χρησιμοποιούμε τον KNN με 3 γείτονες για να πάρουμε μετρήσεις. Το πρόγραμμα που υλοποιείται παίρνει μετρήσεις για 2-39 folds.

ΆΛΛΕΣ ΤΕΧΝΙΚΕΣ ΒΕΛΤΙΣΤΟΠΟΙΗΣΗΣ ΤΗΣ ΑΚΡΙΒΕΙΑΣ

Ύστερα από έρευνα γύρω από την απόδοση του KNN, βλέπουμε πως υπάρχουν και άλλες τεχνικές αύξησης της ακρίβειας του κατηγοριοποιητή. Εκτός από την Cross Validation, που αφορά στην υλοποίηση της κατηγοριοποίησης, μπορούμε να επέμβουμε και στο ίδιο το dataset και να το επεξεργαστούμε πριν το χρησιμοποιήσουμε στον αλγόριθμό μας. Μπορούμε να απομακρύνουμε το θόρυβο χρησιμοποιώντας denoising autoencoders, ή να κάνουμε blur (defocus, linear horizontal motion blur). Ωστόσο αυτές οι τεχνικές απλώς ερευνήθηκαν, χωρίς να υλοποιηθούν στη συγκεκριμένη εργασία.

RBF NEURAL NETWORK

Για το RBFNN αρχικά εισάγουμε τις απαραίτητες βιβλιοθήκες όπως keras, tensorflow, sklearn time και numpy. Στη συνέχεια φορτώνουμε το MNIST Dataset, το αναθέτουμε σε μεταβλητές, το κανονικοποιούμε (normalize) και το κάνουμε reshape. Ύστερα δηλώνουμε τις κλάσεις που θα χρησιμοποιήσουμε για την εκπαίδευση του μοντέλου, όπως την InitCentersRandom, την InitCenterMeans και την RBFLayer. Δημιουργούμε το μοντέλο και το αρχικοποιούμε δίνοντάς του ως όρισμα μια είσοδο. Ζητάμε να μας τυπώσει το σχήμα της εισόδου. Κάνουμε compile το μοντέλο, fit & predict. Τέλος ζητάμε από αυτό να μας τυπώσει διάφορα αποτελέσματα που σχετίζονται με την ακρίβεια και το χρόνο.

Με τη μέθοδο InitCentersRandom, τα κέντρα των ομάδων προκύπτουν τυχαία. Αντίθετα με την μέθοδο InitCenterMeans έχουμε ομαδοποίηση των κέντρων. Αρχικά αναθέτουμε τυχαία τα κέντρα των ομάδων $K_1,K_2...K_n$. Για κάθε x_i υπολογίζουμε το κοντινότερο κέντρο K_i, και το αναθέτουμε στην συστάδα. Στη συνέχεια για κάθε συστάδα $K_1,K_2...K_n$ υπολογίζουμε τα νέα γεωμετρικά τους κέντρα χρησιμοποιώντας τους μέσους όρους απο όλα τα σημεία κάθε συστάδας που προέκυψαν από το προηγούμενο βήμα. Η διαδικασία σταματάει όταν δεν έχουμε μεταβολές στις συστάδες δηλαδή όταν όλα τα στοιχεία παραμένουν ίδια. Αφού ολοκληρωθεί η διαδικασία, μεταβιβάζουμε την πληροφορία στο RBFLayer και εφαρμόζουμε τον αλγόριθμο βελτιστοποίησης RMSProp (Root Mean Square Propagation), στον οποίο πρακτικά διαιρείται ένα δεδομένο βάρος με το κινητό μέσο όρο των τελευταίων κλίσεων για αυτό το βάρος.

AUTOENCODER

Για αυτό τον τύπο νευρωνικού δικτύου, αρχικά εισάγουμε τις απαραίτητες βιβλιοθήκες. Στη συνέχεια υλοποιούμε αρχικά το δίκτυο του κωδικοποιητή, προσθέτοντας στρώσεις (layers – convolutional2D & maxPooling2D) δηλώνοντας τη συνάρτηση ενεργοποίησης και το σχήμα εισόδου. Αντίστοιχα υλοποιούμε το δίκτυο του αποκωδικοποιητή και κάνουμε compile το μοντέλο. Ύστερα φορτώνουμε το dataset MNIST, το κανονικοποιούμε, το κάνουμε reshape και τελικά κάνουμε fit το μοντέλο για 15 epochs & batch size 128.

Γραφική αναπαράσταση μετρήσεων

Figure 1: Predict time of each model

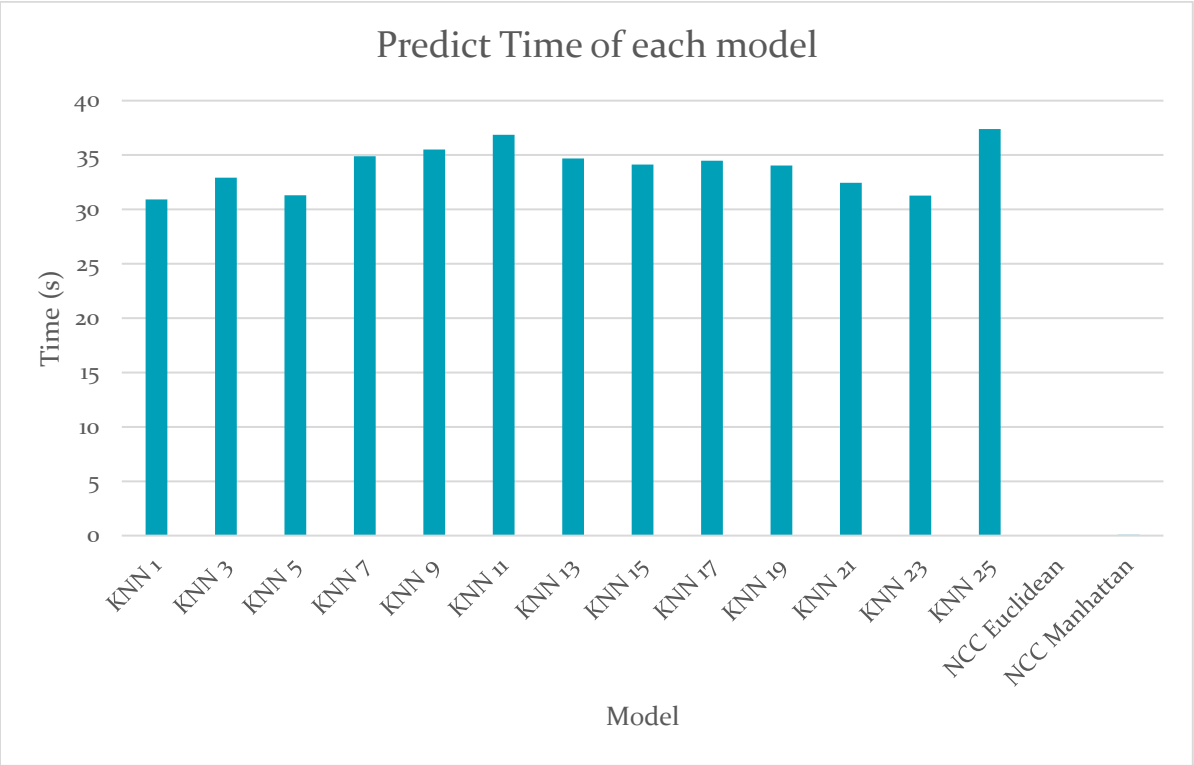


Figure 2: Fit time of each model

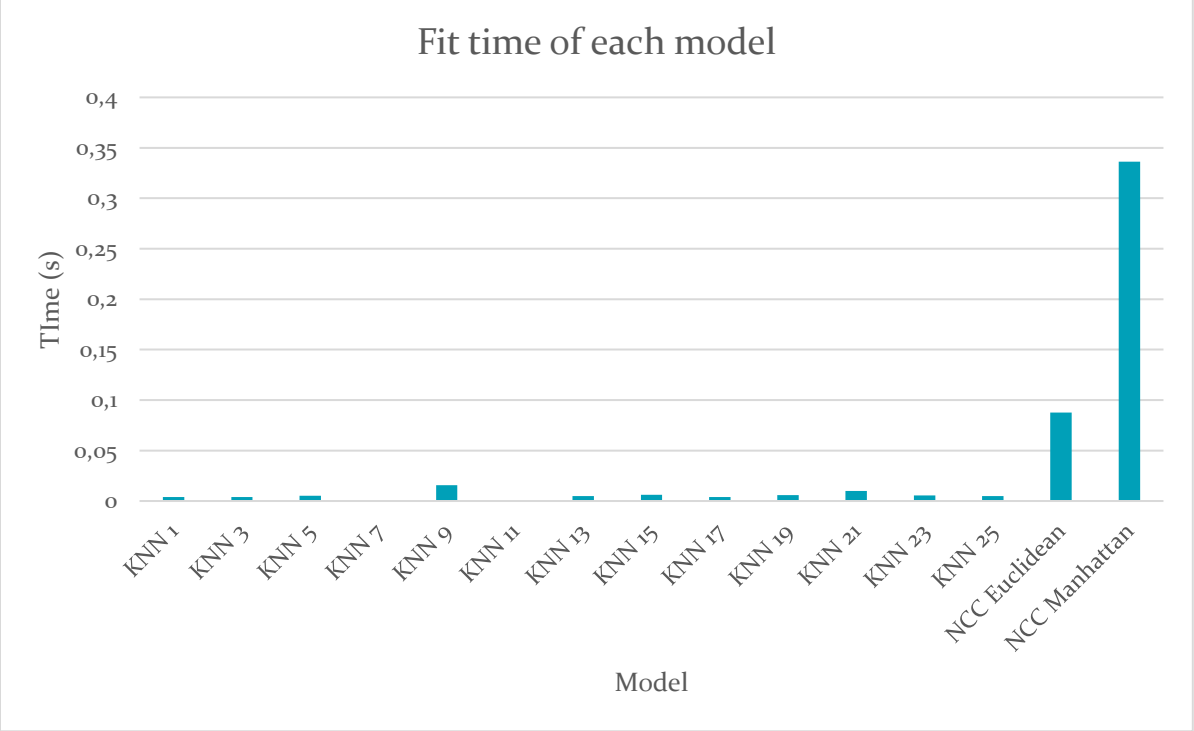


Figure 3: Total time of each model

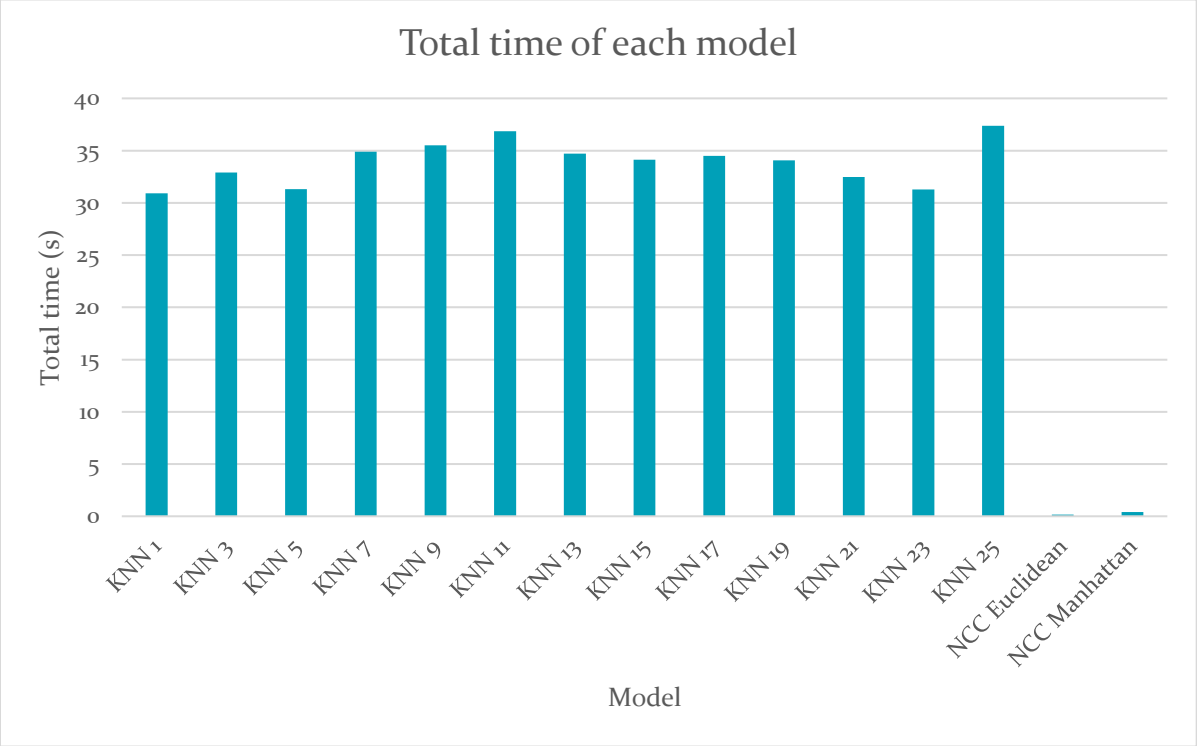


Figure 4: Accuracy percentage of KNN models

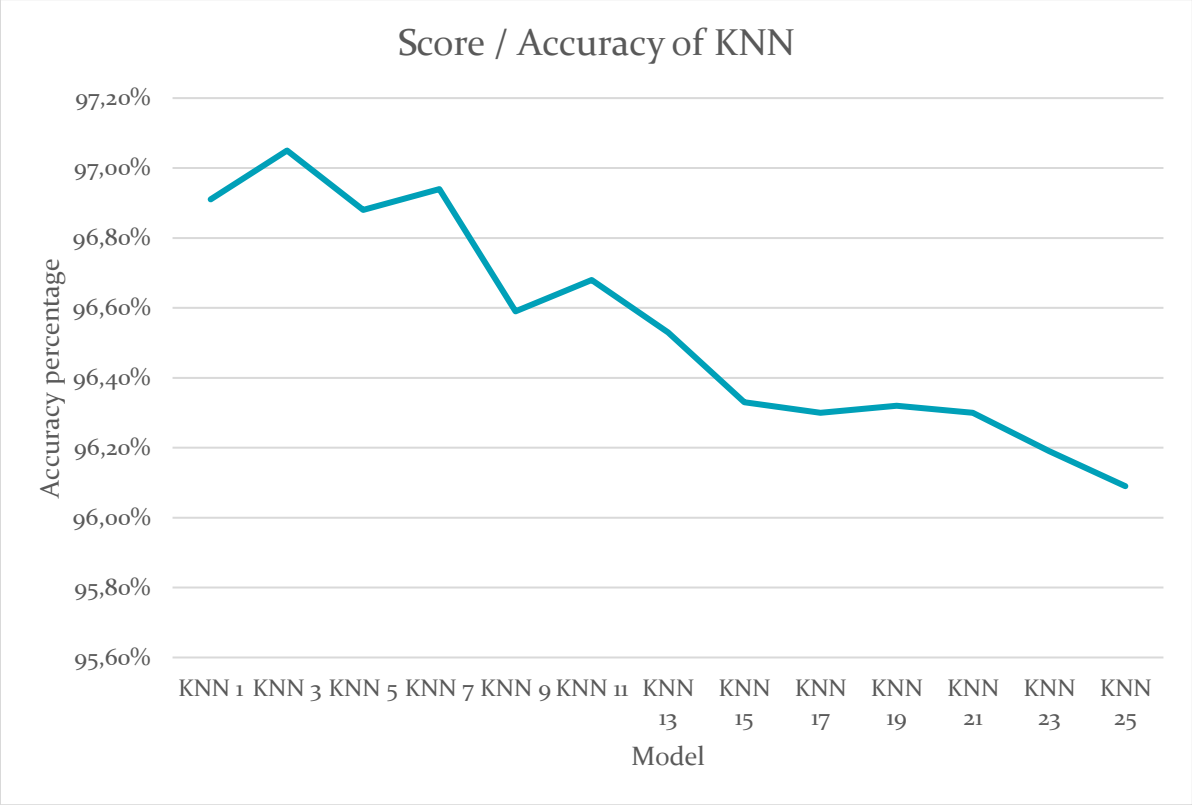


Figure 5: Accuracy percentage of each model

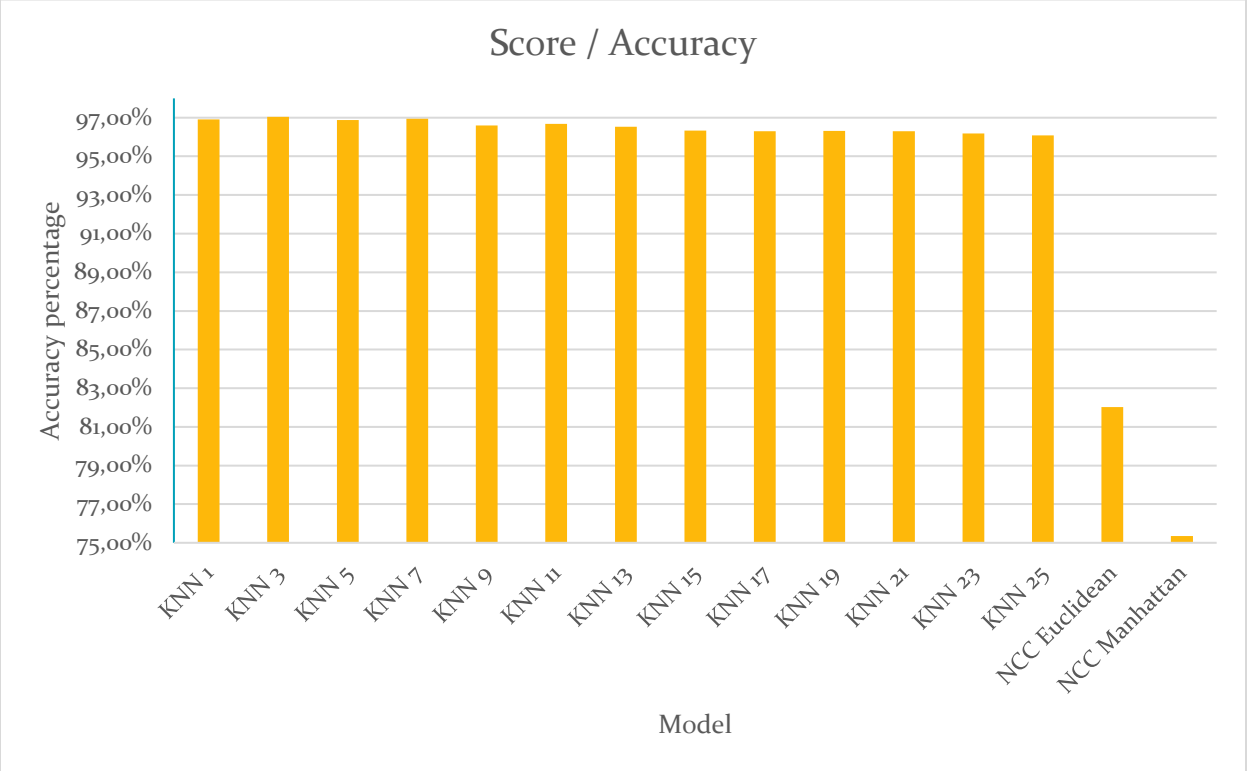


Figure 6: Accuracy percentage of KNN (3 neighbors) with Cross Validation

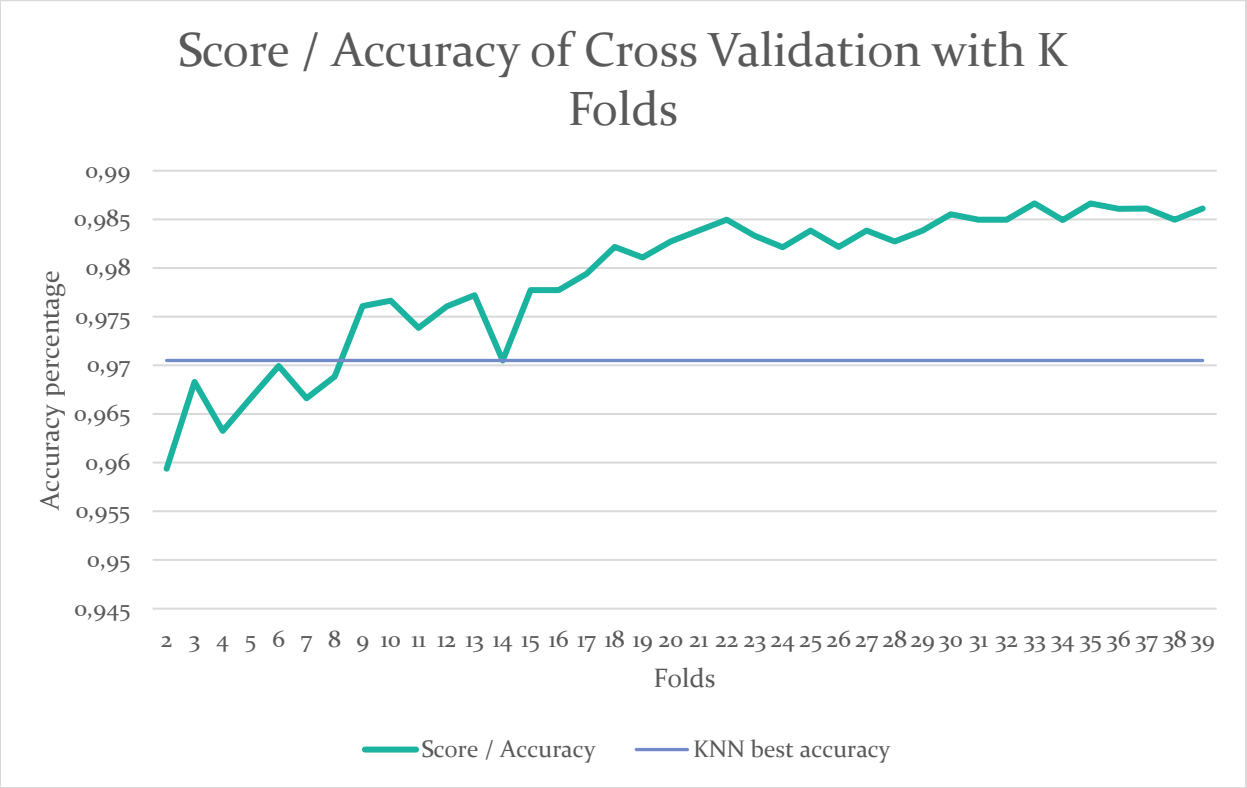


Figure 7: Centers Random Initializer Accuracy vs Epochs with 10, 30 & 100 neurons

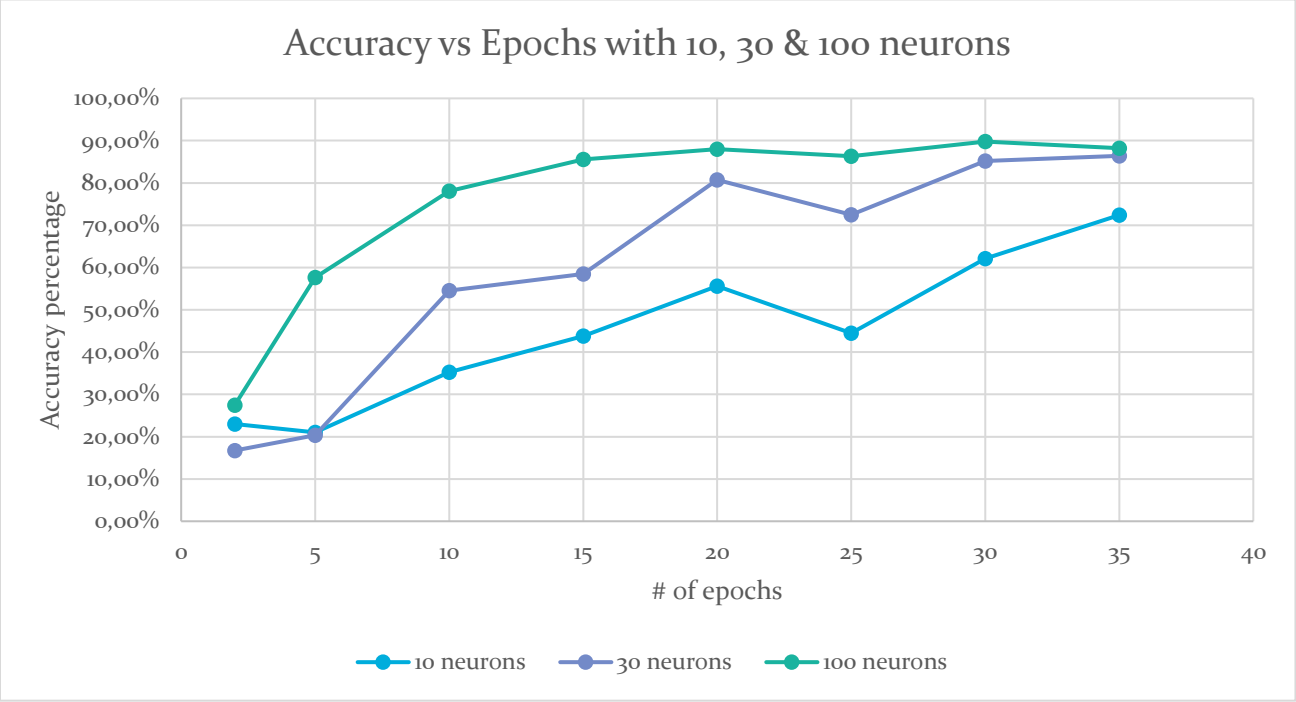


Figure 8: Centers Random Initializer training time vs epochs with 10, 30 & 100 neurons

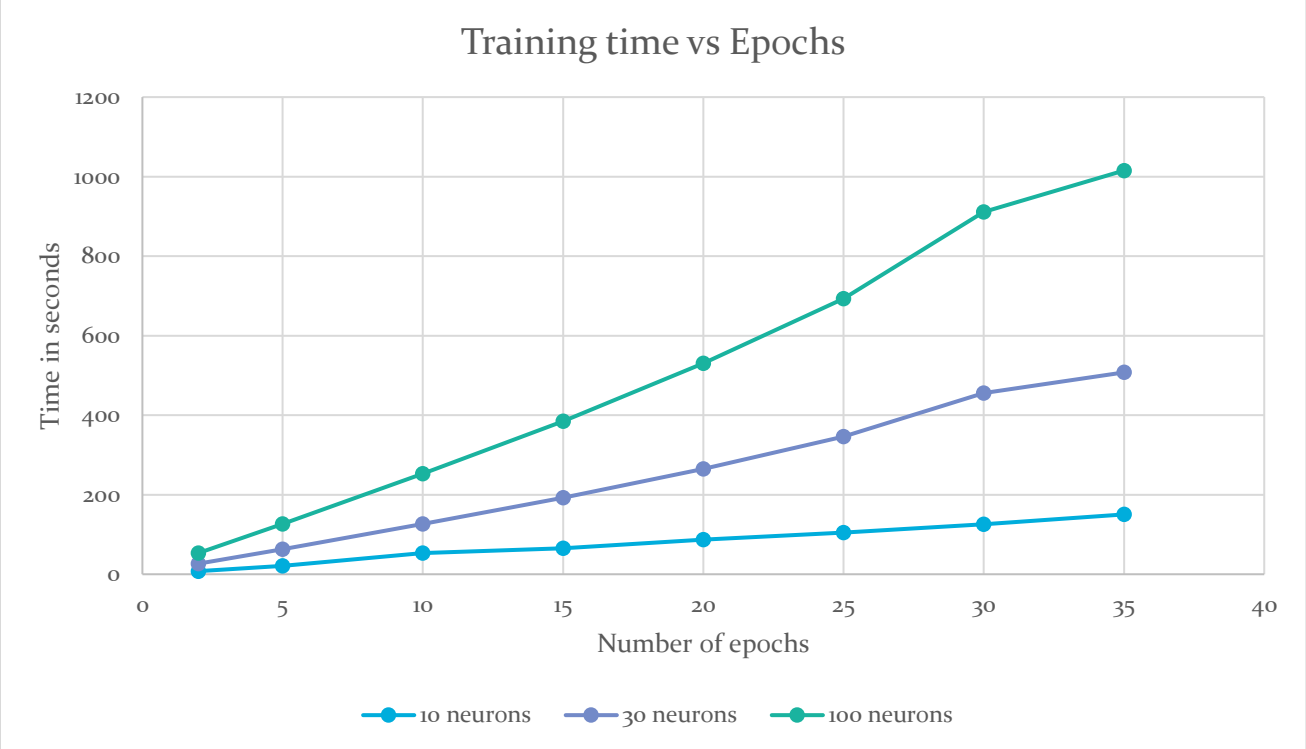


Figure 9: Total params with 10, 30 & 100 neurons

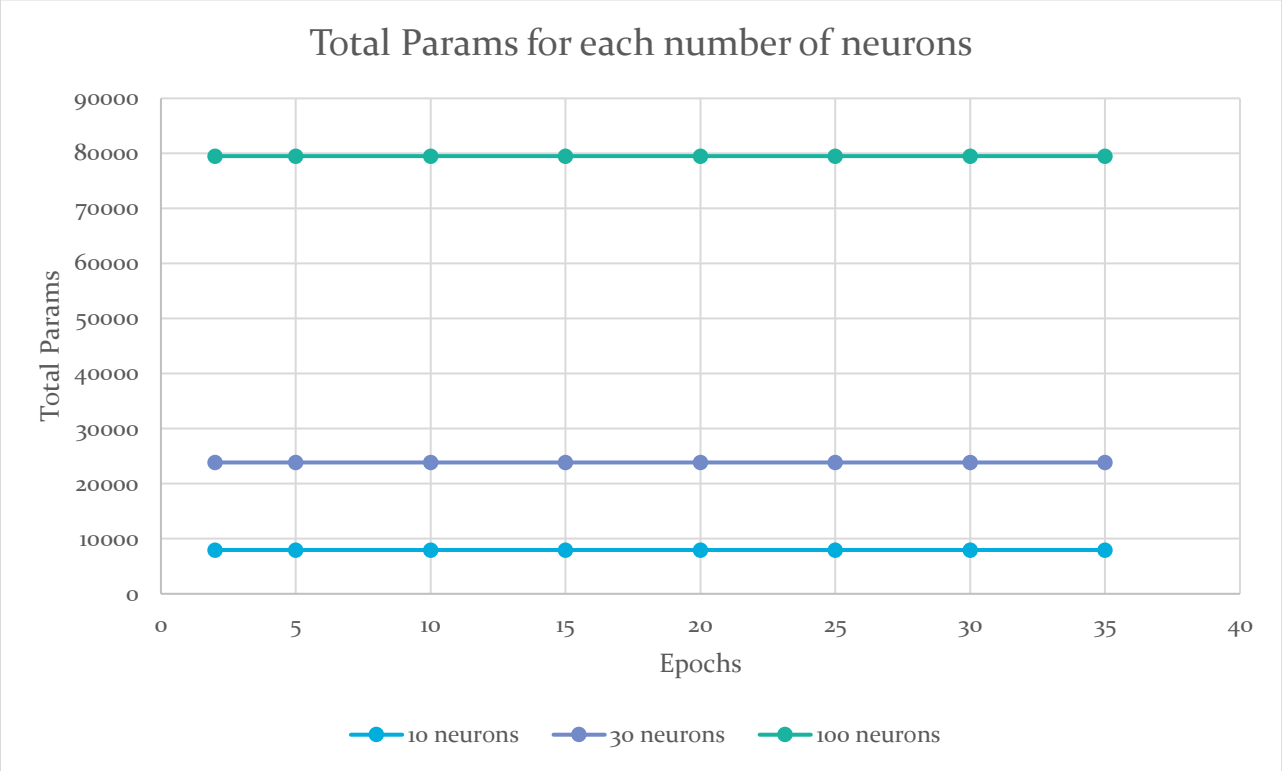


Figure 10: Accuracy vs gamma value with number 10, 30 & 100 neurons in Random Initialization

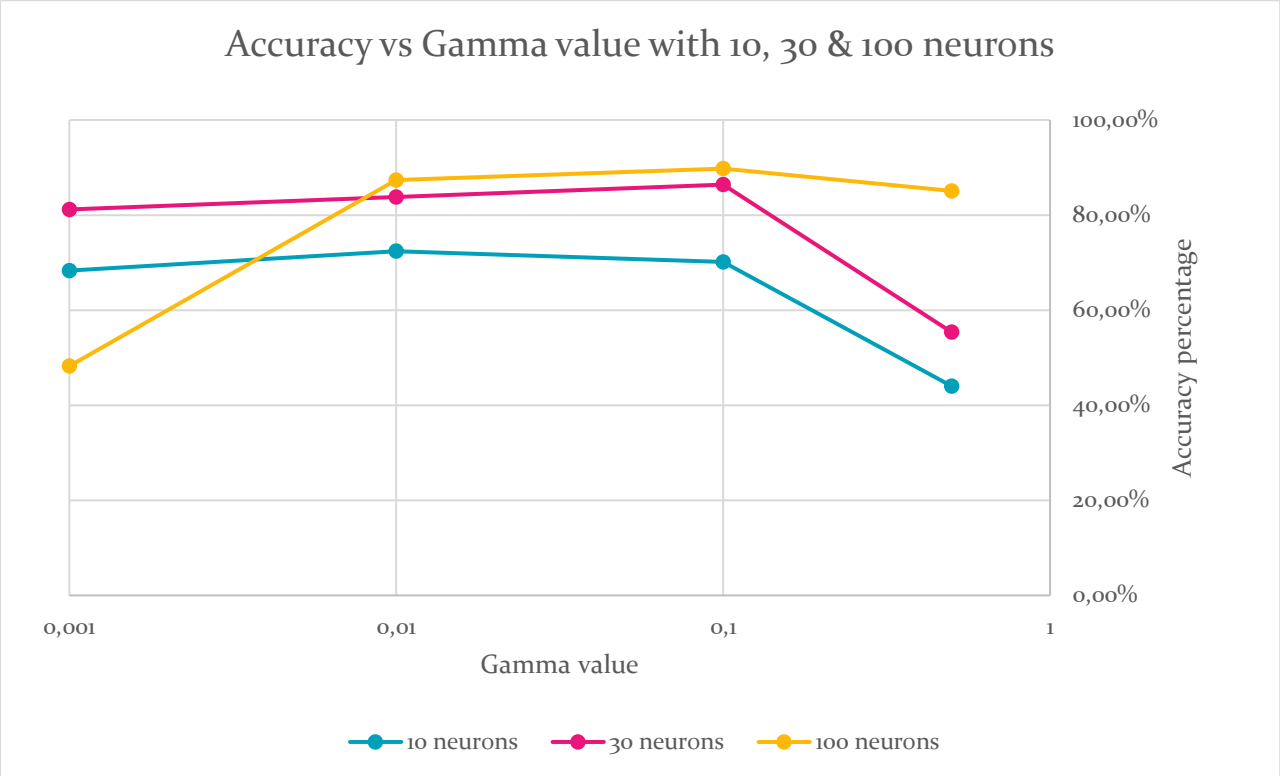


Figure 11: Accuracy vs gamma value with number 10, 30 & 100 neurons in KMeans

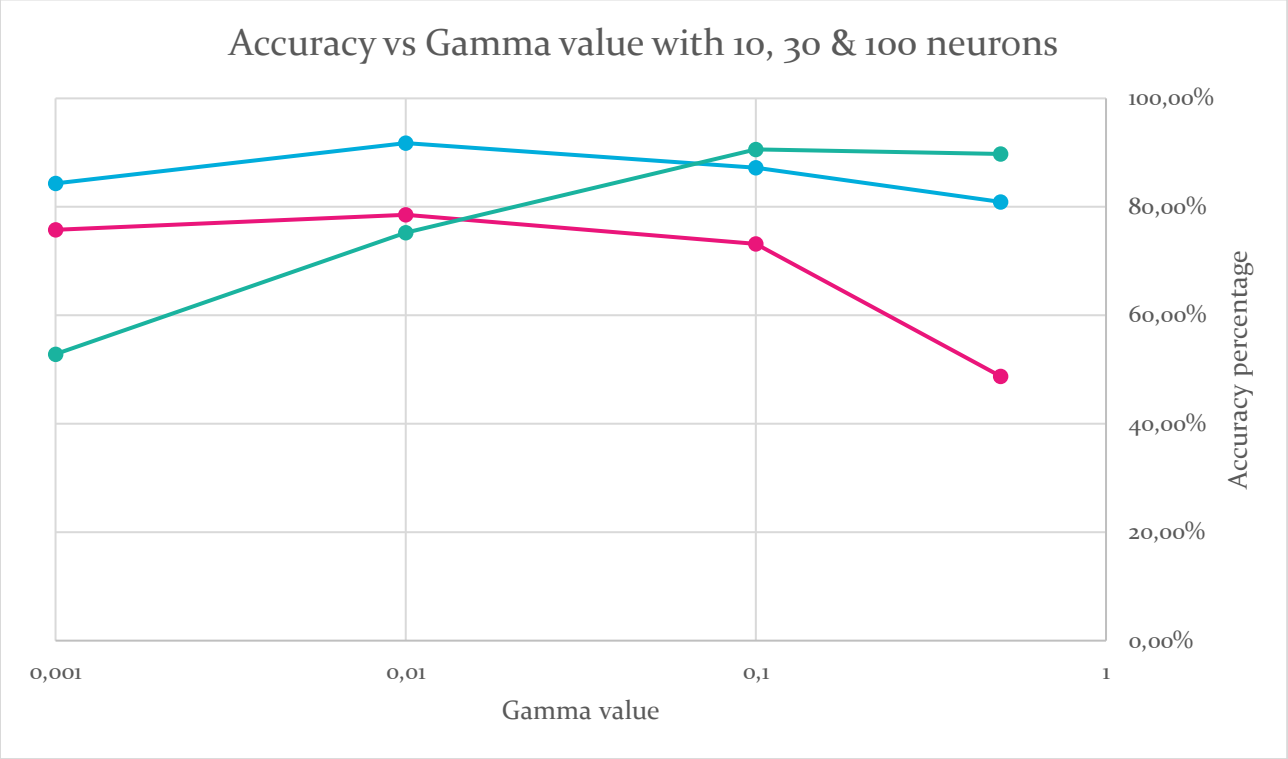
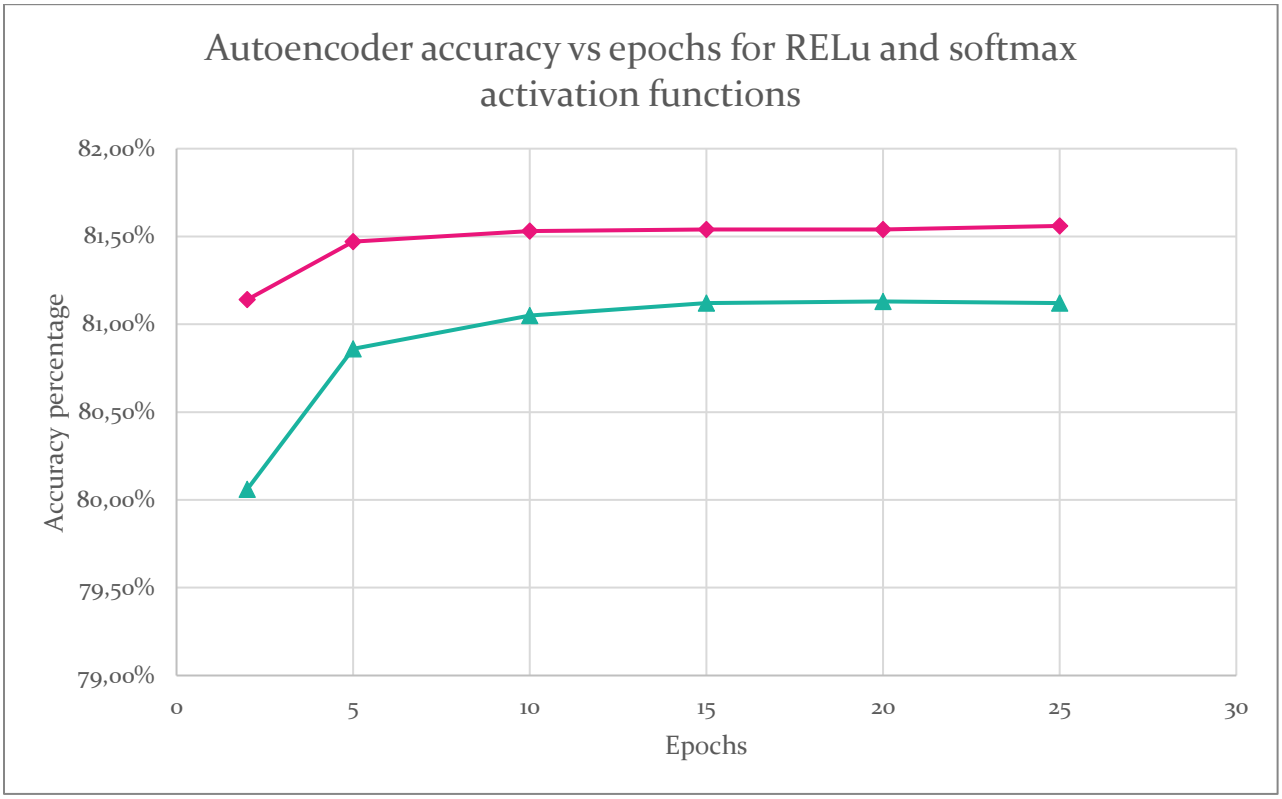


Figure 12: Autoencoder accuracy vs number of epochs



Σχολιασμός Αποτελεσμάτων

KNN VS NCC METRICS

Συγκρίνοντας τα αποτελέσματα μεταξύ των KNN με διαφορετικό αριθμό γειτόνων, παρατηρούμε στο γράφημα 4 πως πιο ακριβής είναι αυτός με τους 3 γείτονες (97,05%). Ο συνολικός χρόνος κυμαίνεται από 31 μέχρι 37 δευτερόλεπτα, επομένως θεωρείται γενικά αργός αλγόριθμος. Να σημειωθεί πως ο χρόνος ποικίλλει ανάλογα με την επεξεργαστική ισχύ του μηχανήματος που χρησιμοποιούμε. Ο ίδιος κώδικας έτρεξε και σε διαφορετικό μηχάνημα και σε διαφορετικό περιβάλλον, και αυτό είχε σημαντική επιρροή στο χρόνο εκπαίδευσης και πρόβλεψης. Ωστόσο, όλες οι μετρήσεις που φαίνονται στους πίνακες του παραρτήματος Α λήφθηκαν ceteris paribus, και επομένως η σύγκριση μεταξύ των αποτελεσμάτων έχει νόημα.

Ο NCC από την άλλη παρατηρούμε πως είναι αρκετά γρήγορος. Στη περίπτωσή μας έχουμε κάνει reshape(flatten) το dataset από 2D σε 1D (28 * 28 to 784), επομένως έχει νόημα η ευκλείδεια απόσταση. [Γενικά δεδομένα σε 1D χρησιμοποιούνται σε πλήρως συνδεδεμένα νευρωνικά δίκτυα (όπως MLP), ενώ δεδομένα σε 2D χρησιμοποιούνται για συνελικτικά νευρωνικά δίκτυα.] Για σκοπούς παρατήρησης, υλοποιήθηκε ο NCC και με Manhattan απόσταση, ωστόσο αυτό είχε σημαντική επίδραση (μείωση) στην ακρίβειά του.

Παρατηρούμε επίσης στα γραφήματα 1 και 2 πως ο KNN έχει αρκετά μικρό χρόνο fit και πολύ μεγαλύτερο predict σε σύγκριση με τον NCC. Η μεγάλη διαφορά που βλέπουμε στους χρόνους μεταξύ KNN και NCC έγκειται στο γεγονός πως η περισσότερη δουλειά στον NCC γίνεται κατά τη διαδικασία fit, όπου ο αλγόριθμος υπολογίζει τα κέντρα (centroids) των κλάσεων, και επομένως στη συνέχεια απλώς συγκρίνει τις αποστάσεις αυτών από το σημείο που μας ενδιαφέρει. Δηλαδή στη συγκεκριμένη περίπτωση υλοποιεί 10 συγκρίσεις αφού έχουμε 10 κλάσεις. Ο KNN κάθε φορά ψάχνει τους πλησιέστερους K γείτονες και υπολογίζει πόσοι από αυτούς ανήκουν στην κάθε κλάση, ώστε να κατηγοριοποιήσει το σημείο που μελετάμε. Για το λόγο αυτό παρατηρούμε και μια μικρή αύξηση στο χρόνο όσο η παράμετρος K αυξάνεται.

Τελικά, ο NCC είναι σαφώς πιο γρήγορος από τον KNN. Ωστόσο, η ακρίβεια των προβλέψεων του NCC είναι αρκετά πιο χαμηλή (στην καλύτερη περίπτωση 82,03% έναντι 97,05% του KNN).

CROSS VALIDATION VS KNN METRICS

Χρησιμοποιήσαμε τη τεχνική Cross Validation στον KNN με 3 γείτονες καθώς αυτός είχε τη μεγαλύτερη ακρίβεια στη προηγούμενη σύγκριση. Βλέπουμε, από το γράφημα 6, πως η ακρίβεια αυξάνεται για μεγαλύτερα K-Folds, ενώ για K=35 φτάνουμε στην μέγιστη ακρίβεια που παρατηρήθηκε, 0,98664 ή 98,664%. Ακόμη όμως παρατηρούμε πως με τη τεχνική CV, οι περισσότερες μετρήσεις μας έχουν ακρίβεια μεγαλύτερη από 97,05%, η οποία ήταν η μέγιστη στο μοντέλο KNN χωρίς CV. Τελικά, συμπεραίνουμε πως με τη τεχνική CV, η ακρίβεια στα αποτελέσματά μας είναι σαφώς καλύτερη, ειδικά όσο αυξάνουμε τα folds.

RBFNN METRICS

Η ακρίβεια του RBFNN δεν φτάνει σε μεγάλα επίπεδα. Δοκιμάσαμε να εκπαιδεύσουμε το δίκτυο με 2 αλγορίθμους, αυτόν της τυχαίας αρχικοποίησης κέντρων, και αυτόν της ομαδοποίησης. Η μεγαλύτερη ακρίβεια επιτεύχθηκε μέσω της τυχαίας επιλογής κέντρων, ωστόσο με μικρή διαφορά. Ακόμη δοκιμάσαμε να αλλάξουμε τον αριθμό των κρυφών νευρώνων, τον αριθμό των epochs καθώς και υπερπαραμέτρους όπως το γ (gamma). Παρατηρήσαμε μεταβολές τόσο στο χρόνο και την ακρίβεια, όσο και στον συνολικό αριθμό παραμέτρων. Στον αντίστοιχο πίνακα φαίνονται τα αποτελέσματα των μετρήσεων, ενώ στα αντίστοιχα γραφήματα μπορούμε να δούμε πως μεταβάλλεται η ακρίβεια και ο χρόνος ανάλογα με τον αλγόριθμο εκπαίδευσης και τον αριθμό των epochs. Οι μεγαλύτερες ακρίβειες παρατηρήθηκαν για gamma = 0,01-0,1 ανάλογα με τον αριθμό των νευρώνων στο κρυφό επίπεδο. Οι μέση ακρίβεια ήταν υψηλότερη χρησιμοποιώντας τον αλγόριθμο InitCentersKmeans για την εκπαίδευση του δικτύου. Η υψηλότερη τιμή που καταγράφηκε ήταν 91,75% για 100 νευρώνες στο κρυφό επίπεδο, για 15 epochs & gamma = 0,01.

AUTOENCODER METRICS

Στον αυτοκωδικοποιητή η ακρίβεια δεν έφτασε σε εξαιρετικά επίπεδα. Δοκιμάσαμε να αλλάξουμε τον αριθμό από epochs, τα ορίσματα των convolutional & maxpooling 2D layers (όπως τη συνάρτηση ενεργοποίησης). Στην ακρίβεια παρατηρήθηκαν πολύ μικρές διαφορές, μικρότερες της τάξης του 1%. Παρατηρώντας επίσης την ακρίβεια για κάθε epoch χωριστά, βλέπουμε πως μετά από κάποιον αριθμό epochs (περίπου 7), η ακρίβεια του μοντέλου φτάνει σε κορεσμό, και από εκεί και πέρα παραμένει σταθερή. Για το λόγο αυτό παρατηρείται και μια γραμμικότητα στο αντίστοιχο γράφημα, σε σύγκριση με άλλα μοντέλα που τα ποσά είναι ανάλογα, και για μεγαλύτερο αριθμό epochs παρατηρείται αύξηση της ακρίβειας.

KNN, NCC, KNN CV, RBFNN VS AUTOENCODER METRICS

Συγκρίνοντας όλους τους κατηγοριοποιητές, καταλήγουμε πως ο πιο ακριβής από όλους ήταν ο KNN. Ο χρόνος fit είναι αρκετά μικρότερος στο μοντέλο KNN. Ο NCC κερδίζει σε χρόνο predict και τους δύο προηγούμενους κατηγοριοποιητές, ωστόσο χάνει γενικά σε ακρίβεια. Το RBFNN δεν έχει πολύ μεγάλη ακρίβεια, ωστόσο με μικρό αριθμό νευρώνων στο κρυφό επίπεδο και μικρό αριθμό epochs είναι αρκετά γρήγορος, με χρόνο μόλις λίγα δευτερόλεπτα. Τέλος η ακρίβεια του αυτοκωδικοποιητή φτάνει σε κορεσμό, με ποσοστό περίπου 81%.

Παράρτημα

(Α) ΠΙΝΑΚΕΣ ΜΕΤΡΗΣΕΩΝ

Table 1: Μετρήσεις fit time και predict time KNN & NCC

Model	Fit time (s)	Predict time (s)
KNN 1	0,004010677337646484	30,921151638031006
KNN 3	0,003977298736572266	32,90735363960266
KNN 5	0,0053746700286865234	31,2930598258972l7
KNN 7	0,0	34,9021737575531
KNN 9	0,01564168930053711	35,497914791107l8
KNN 11	0,0	36,85045766830444
KNN 13	0,004988431930541992	34,698371887207O3
KNN 15	0,006287336349487305	34,1180405616760l5
KNN 17	0,003983736038208008	34,49005627632141
KNN 19	0,0057260990142822266	34,0493946075439l5
KNN 21	0,010051488876342773	32,45123600959778
KNN 23	0,0055880546569824l2	31,26777744293213
KNN 25	0,004971265792846l8	37,37762904167175
NCC Euclidean	0,08776473999023438	0,06194138526916504
NCC Manhattan	0,3364279270172119	0,07768750190734863

Table 2: Μετρήσεις συνολικού χρόνου και ακρίβειας KNN & NCC

Model	Total time (s)	Score / Accuracy
KNN 1	30,9251623153686 52	0,9691 or 96,91%
KNN 3	32,91133094	0,9705 or 97,05%
KNN 5	31,2984345	0,9688 or 96,88%
KNN 7	34,90217376	0,9694 or 96,94%
KNN 9	35,51355648	0,9659 or 96,59%
KNN 11	36,85045767	0,9668 or 96,68%
KNN 13	34,70336032	0,9653 or 96,53%
KNN 15	34,1243279	0,9633 or 96,33%
KNN 17	34,49404001	0,963 or 96,3%
KNN 19	34,05512071	0,9632 or 96,32%
KNN 21	32,4612875	0,963 or 96,3%
KNN 23	31,2733655	0,9619 or 96,19%
KNN 25	37,3826003074646	0,9609 or 96,09%
NCC Euclidean	0,14970612525939942	0,8203 or 82,03%
NCC Manhattan	0,414115428924560 53	0,7535 or 75,35%

Table 3: Μετρήσεις ακρίβειας τεχνικής Cross Validation σε KNN Classifier με 3 γείτονες

Folds	Score / Accuracy
2	0,9593788941437034
3	0,9682804674457429
4	0,9632764167285326
5	0,966621788919839
6	0,9699498327759198
7	0,9666155676764869
8	0,9688492063492065
9	0,9760831937465103
10	0,9766325263811299
11	0,9738651667052085
12	0,9760700969425803
13	0,9771941644009209
14	0,9705019725913621
15	0,9777170868347338
16	0,9777328934892543
17	0,9794091221394217
18	0,982182940516274
19	0,9810868155831909
20	0,9827340823970039
21	0,9838512149045665
22	0,9849648243957188

23	0,983312871315073
24	0,9821471471471472
25	0,9838341158059468
26	0,9821866539257845
27	0,9838350335862772
28	0,9827352335164835
29	0,9838618501431463
30	0,9855084745762712
31	0,9849735573639326
32	0,9849819862155391
33	0,9866340169370473
34	0,9849419448476053
35	0,9866408101702218
36	0,986077097505669
37	0,9861072807501378
38	0,9849640724150803
39	0,9861120994330986

Table 4: Best accuracy vs number of epochs for Random Centers Initialization Algorithm

Number of epochs	Accuracy 10 neurons	Accuracy 30 neurons	Accuracy 100 neurons
2	16,73%	16,73%	27,48%
5	20,31%	20,31%	57,66%
10	54,54%	54,54%	78,07%
15	58,45%	58,45%	85,60%
20	80,76%	80,76%	87,98%
25	72,51%	72,51%	86,30%
30	85,22%	85,22%	89,78%
35	86,41%	86,41%	88,23%

Table 5: Best accuracy vs number of epochs for KMeans Clustering Initialization Algorithm

Number of epochs	Accuracy 10 neurons	Accuracy 30 neurons	Accuracy 100 neurons
2	17,49%	19,67%	34,74%
5	22,36%	27,36%	61,34%
10	51,38%	49,85%	82,62%
15	66,43%	78,53%	91,75%
20	69,62%	70,76%	91,67%
25	81,59%	72,51%	91,69%
30	83,97%	78,51%	90,96%
35	90,59%	76,49%	91,54%

(Β) ΚΩΔΙΚΑΣ

KNN & NCC

```python
# import the libraries
from keras.datasets import mnist
from sklearn.neighbors import KNeighborsClassifier, NearestCentroid
from time import time
import numpy as np

# load the dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
print(f"Train shape: {x_train.shape} ")
print(f"Test shape: {x_test.shape} ")

# reshape model to work with (flatten it into 1D)
x_train = x_train.reshape(x_train.shape[0], x_train.shape[1]**2)
x_test = x_test.reshape(x_test.shape[0], x_test.shape[1]**2)
print(f"Reshaped train shape: {x_train.shape} ")
print(f"Reshaped test shape: {x_test.shape} ")

```

```python
# KNN (for K values: 1 to 25)
kValues = np.arange(1, 27, 2)
scores = []
times = []
for i in kValues:
    knn = KNeighborsClassifier(n_neighbors=i)
    startKNNFitTime = time()
    knn.fit(x_train, y_train)
    fitKNNTime = time()-startKNNFitTime
    accuracyKNN = knn.score(x_test, y_test)
    startKNNPredictTime = time()
    knn.predict(x_test)
    predictKNNTime = time()-startKNNPredictTime
    totalKNNTime = fitKNNTime + predictKNNTime
    times.append(totalKNNTime)
    scores.append(accuracyKNN)
    print(f"KNN with {i} neighbor(s) fit time: {fitKNNTime}s \n"
          f"KNN with {i} neighbor(s) predict time: {predictKNNTime}s \n"
          f"KNN with {i} neighbor(s) accuracy: {accuracyKNN} \n")
    print(times)
    print(scores)
```

```python
# NCC (for distances: Euclidean, Manhattan)
distances = ["manhattan", "euclidean"]
for distance in distances:
    nc = NearestCentroid(distance)
    startNCCFitTime = time()
    nc.fit(x_train, y_train)
    fitNCCTime = time()-startNCCFitTime
    accuracyNCC = nc.score(x_test, y_test)
    startNCCPredictTime = time()
    nc.predict(x_test)
    predictNCCTime = time()-startNCCPredictTime
    totalKNNTime = fitNCCTime + predictNCCTime
    print(f"NCC {distance} fit time: {fitNCCTime}s \n"
          f"NCC {distance} predict time: {predictNCCTime}s \n"
          f"NCC {distance} accuracy: {accuracyNCC} \n")
```

KNN Cross Validation

```python
# import the libraries
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split, KFold, StratifiedKFold, cross_val_score

# load the dataset
digits = load_digits()

# split the dataset
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target)

# for 3 neighbors in KNN, for K fold values 2-40 print the mean accuracy
for i in range(2, 40):
    print(sum(cross_val_score(KNeighborsClassifier(n_neighbors=3), digits.data, digits.target, cv=i))/i)
```

RBFNN

```python
from time import time

import keras
import numpy as np
from keras.datasets import mnist
from keras.engine.base_layer_v1 import Layer
from keras.initializers.initializers_v1 import RandomUniform
from keras.initializers.initializers_v2 import Initializer, Constant
from keras.layers import Dense
from keras.optimizers import RMSprop
from keras.utils import to_categorical
from tensorflow import reduce_sum
from tensorflow.python.keras.backend import expand_dims, transpose, exp


# Configuration options
feature_vector_length = 784
num_classes = 10

# Load the data
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

# Reshape to 28 x 28 pixels = 784 features
X_train = X_train.reshape(X_train.shape[0], feature_vector_length)
X_test = X_test.reshape(X_test.shape[0], feature_vector_length)

# Standardize dataset
X_train = X_train.astype("float32")
X_test = X_test.astype("float32")
X_train /= 255.
X_test /= 255.

# Convert target classes to categorical ones
Y_train = to_categorical(Y_train, num_classes)
Y_test = to_categorical(Y_test, num_classes)
```

```python

# Set the input shape
input_shape = (feature_vector_length,)
print(f"Feature shape: {input_shape}")


class InitCentersRandom(Initializer):
    # """ Initializer for initialization of centers of RBF network
    #     as random samples from the given data set.

    # Arguments
    # X: matrix, dataset to choose the centers from (random rows
    #   are taken as centers)

    def __init__(self, X):
        self.X = X
        super().__init__()

    def __call__(self, shape, dtype=None):
        assert shape[1:] == self.X.shape[1:]  # check dimension

        # np.random.randint returns ints from [low, high) !
        idx = np.random.randint(self.X.shape[0], size=shape[0])

        return self.X[idx, :]
```

```python
class RBFLayer(Layer):
    def __init__(self, output_dim, initializer=None, betas=1.0, **kwargs):

        self.output_dim = output_dim

        # betas is either initializer object or float
        if isinstance(betas, Initializer):
            self.betas_initializer = betas
        else:
            self.betas_initializer = Constant(value=betas)

        self.initializer = initializer if initializer else RandomUniform(0.0, 1.0)

        super().__init__(**kwargs)

    def build(self, input_shape):

        self.centers = self.add_weight(
            name="centers",
            shape=(self.output_dim, input_shape[1]),
            initializer=self.initializer,
            trainable=True,
        )
        self.betas = self.add_weight(
            name="betas",
            shape=(self.output_dim,),
            initializer=self.betas_initializer,
            # initializer='ones',
            trainable=True,
        )

        super().build(input_shape)
```

```python
    def call(self, x):

        C = expand_dims(self.centers, -1)  # inserts a dimension of 1
        H = transpose(C - transpose(x))  # matrix of differences
        return exp(-self.betas * reduce_sum(H ** 2, axis=1))

    def compute_output_shape(self, input_shape):
        return (input_shape[0], self.output_dim)

    def get_config(self):
        # have to define get_config to be able to use model_from_json
        config = {"output_dim": self.output_dim}
        base_config = super().get_config()
        return dict(list(base_config.items()) + list(config.items()))


# create RBF network as keras sequential model
RBFlayer = RBFLayer(10, initializer=InitCentersRandom(X_train), betas=0.5, input_shape=input_shape)
# RBFlayer = RBFLayer(100, initializer=InitCentersKMeans(X_train), betas=0.1, input_shape=input_shape)

inputs = keras.layers.Input(input_shape)  # input layer
x = RBFlayer(inputs)  # hidden layer
outputs = Dense(10, activation="softmax")(x)  # output layer

model = keras.models.Model(inputs, outputs)
print(input_shape)
print(model.summary())

# compile the model
model.compile(loss="mean_squared_error", optimizer=RMSprop(), metrics=["accuracy"])
```

```python
# fit and predict
start = time()
h = model.fit(X_train, Y_train, batch_size=50, epochs=35, verbose=1, validation_split=0.2)
train_time = time() - start

# Test the model after training
test_results = model.evaluate(X_test, Y_test, verbose=1)
print(f"Test results - Loss: {test_results[0]} - Accuracy: {test_results[1]}")

# Test the model after training
test_results = model.evaluate(X_test, Y_test, verbose=1)
print(f"Test results - Loss: {test_results[0]} - Accuracy: {test_results[1]}")

# accuracy values at the end of each epoch (if you have used `acc` metric)
print("accuracies: ")
print(h.history["accuracy"])

# training time
print("training time: ")
print(train_time)

# list of epochs number
print(h.epoch)

```

AUTOENCODER

```python
from keras.layers import Dense, Conv2D, MaxPooling2D, UpSampling2D
from keras.models import Sequential
from keras import Input, Model
from keras.datasets import mnist
import numpy as np


model = Sequential()

# encoder network
model.add(Conv2D(30, 3, activation='relu', padding='same', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(2, padding='same'))
model.add(Conv2D(15, 3, activation='relu', padding='same'))
model.add(MaxPooling2D(2, padding='same'))

# decoder network
model.add(Conv2D(15, 3, activation='relu', padding='same'))
model.add(UpSampling2D(2))
model.add(Conv2D(30, 3, activation='relu', padding='same'))
model.add(UpSampling2D(2))
model.add(Conv2D(1, 3, activation='sigmoid', padding='same'))

# output layer
model.compile(optimizer='adam', loss='binary_crossentropy')
model.summary()

(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))
model.fit(x_train, x_train, epochs=15, batch_size=128, validation_data=(x_test, x_test))

```

Βιβλιογραφία

1. Ahadli, Tarlan. “Most Effective Way to Implement Radial Basis Function Neural Network for Classification Problem.” Medium, Towards Data Science, 10 Jan. 2020, https://towardsdatascience.com/most-effective-way-to-implement-radial-basis-function-neural-network-for-classification-problem-33c467803319
2. Hubens, Nathan. “Deep inside: Autoencoders.” Medium, Towards Data Science, 10 Apr. 2018, https://towardsdatascience.com/deep-inside-autoencoders-7e41f319999f
3. Mondal, Arnab. “Autoencoders Python: How to Use Autoencoders in Python.” Analytics Vidhya, 29 June 2021, https://www.analyticsvidhya.com/blog/2021/06/complete-guide-on-how-to-use-autoencoders-in-python/
4. Raaaouf. “Raaaouf/RBF_neural_network_python: An Implementation of a Radial Basis Function Neural Network (RBFNN) for Classification Problem.” GitHub, https://github.com/raaaouf/RBF_neural_network_python
5. Team, Keras. “Keras Documentation: Conv2d Layer.” Keras, https://keras.io/api/layers/convolution_layers/convolution2d/.
6. Vidnerova, Petra. “Petravidnerova/RBF_KERAS: RBF Layer for Keras.” GitHub, https://github.com/PetraVidnerova/rbf_keras