Neural Networks
&
Deep Learning

1Η ΥΠΟΧΡΕΩΤΙΚΗ ΕΡΓΑΣΙΑ

Κωνσταντίνος Μυλωνάς | 10027 | 27.11.2022

Word Count: 3310

Πίνακας περιεχομένων

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 πολυστρωματικού perceptron που να εκπαιδεύεται με τον αλγόριθμο back propagation και η σύγκριση της απόδοσης αυτού με τους παραπάνω κατηγοριοποιητές. Το νευρωνικό δίκτυο αυτό εκπαιδεύεται για να διαχωρίζει χειρόγραφα ψηφία. Το πρόβλημα αυτό εμπίπτει στη κατηγορία της επίλυσης προβλήματος κατηγοριοποίησης πολλών κλάσεων (10 στη συγκεκριμένη περίπτωση).

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 πολυστρωματικό (multilayer) perceptron, αποτελεί ένα feedforward τεχνητό νευρωνικό δίκτυο. Απαιτεί τουλάχιστον 3 layers από κόμβους, ένα input, ένα hidden και ένα output, αλλά μπορεί να έχει και παραπάνω από 1 hidden. Κάθε νευρώνας κάθε επιπέδου είναι συνδεδεμένος με κάθε νευρώνα του επόμενου επιπέδου, γι' αυτό και το νευρωνικό δίκτυο είναι πλήρως συνδεδεμένο. Δίνουμε ένα σετ δεδομένων στο επίπεδο εισόδου ως είσοδο και στη συνέχεια αυτά «ζυγίζονται», αποθηκεύουμε το βάρος τους και περνούν ως είσοδος στο επόμενο επίπεδο νευρώνων. Η διαδικασία συνεχίζεται ανάλογα με τα κρυφά επίπεδα νευρώνων που έχουμε, μέχρι να φτάσουμε στο επίπεδο εξόδου (output layer). Όταν εκπαιδεύουμε το δίκτυο με τον αλγόριθμο back propagation πρακτικά έχουμε δύο στάδια. Το πρώτο είναι το feedforward phase, στο οποίο το σήμα εισόδου περνά μέσα από τα κρυφά επίπεδα του δικτύου και τροποποιείται ανάλογα με τα βάρη και τις πολώσεις των νευρώνων και των συνδέσεων, καθώς και από τις συναρτήσεις ενεργοποίησης. Από τη διαδικασία αυτή προκύπτουν τιμές εξόδου, οι οποίες συγκρίνονται με τις επιθυμητές τιμές εξόδου (target outputs) μέσω της συνάρτησης loss. Στη συνέχεια έχουμε το feedback phase, στο οποίο το σήμα σφάλματος περνάει προς την αντίθετη κατεύθυνση στο νευρωνικό δίκτυο, τροποποιώντας τα βάρη των νευρώνων ώστε να έχουμε το ελάχιστο δυνατό σφάλμα.

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) 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

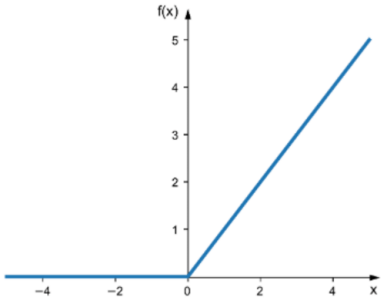
Ύστερα από έρευνα γύρω από την απόδοση του KNN, βλέπουμε πως υπάρχουν και άλλες τεχνικές αύξησης της ακρίβειας του κατηγοριοποιητή. Εκτός από την Cross Validation, που αφορά στην υλοποίηση της κατηγοριοποίησης, μπορούμε να επέμβουμε και στο ίδιο το dataset και να το επεξεργαστούμε πριν το χρησιμοποιήσουμε 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.

MULTI-LAYER PERCEPTRON (DEEP LEARNING ARCHITECTURE)

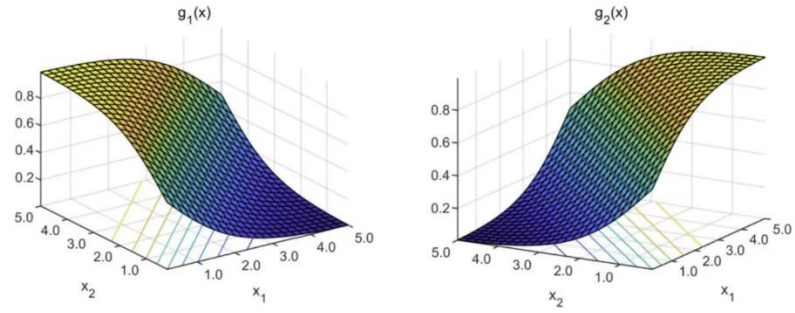
Σε αυτό το κομμάτι της εργασίας υλοποιούμε ένα πολυστρωματικό νευρωνικό δίκτυο perceptron (multilayer perceptron), στο εξής MLP. Χρησιμοποιούμε deep learning αρχιτεκτονική. Κάνουμε reshape το dataset σε μονοδιάστατο διάνυσμα μεγέθους 784 στοιχείων, με τύπο float32, και το κανονικοποιούμε μεταξύ 0 και 1. Στη συνέχεια εισάγουμε 2 κρυφά επίπεδα νευρώνων (256 και 128) με συνάρτηση ενεργοποίησης τη "ReLU". Το output layer έχει 10 νευρώνες, καθώς 10 είναι και οι κλάσεις που θέλουμε να διαχωρίσουμε τα ψηφία (από 0 μέχρι 9), και συνάρτηση ενεργοποίησης την "softmax", η οποία χρησιμοποιείται ευρέως στα επίπεδα εξόδου, καθώς μας δίνει την κατανομή πιθανότητας για τα πιθανά αποτελέσματά μας. Ύστερα κάνουμε compile το μοντέλο μας, χρησιμοποιώντας τον adam optimizer και τη συνάρτηση loss: categorical cross entropy, η οποία χρησιμοποιείται σε συνδυασμό με την συνάρτηση ενεργοποίησης softmax που χρησιμοποιήσαμε στο επίπεδο εξόδου του MLP. Τέλος, κάνουμε fit το μοντέλο, θέτοντας τον αριθμό των epochs σε 100, και το batch size σε 128. Σαφώς αυτές οι παράμετροι μπορούν να αλλάξουν, ωστόσο έδωσαν επαρκή αποτελέσματα για τους σκοπούς της συγκεκριμένης εργασίας.

Παρακάτω φαίνονται οι γραφικές παραστάσεις των συναρτήσεων που αναφέρθηκαν παραπάνω.

Picture 1: ReLU Function



Picture 2: Softmax Function



MULTI-LAYER PERCEPTRON (WITH BACK PROPAGATION)

Επίσης, έγινε προσπάθεια υλοποίησης του νευρωνικού δικτύου χωρίς να πάρουμε έτοιμο το μοντέλο. Υλοποιήσαμε επομένως τον κώδικα που φαίνεται στο αντίστοιχο παράρτημα. Αρχικά εισάγουμε τις απαραίτητες βιβλιοθήκες, και στη συνέχεια δημιουργούμε το MLP ως μια κλάση, μέσα στην οποία ορίζουμε τις παρακάτω μεθόδους:

- μέθοδος init: η συνάρτηση αυτή αρχικοποιεί το δίκτυο με τους βάση τους αριθμούς που δίνουμε ως όρισμα. Δημιουργεί τον απαραίτητο αριθμό από επίπεδα νευρώνων και αρχικοποιεί τα βάρη των ακμών μεταξύ τους,
- μέθοδος forward propagate: η συνάρτηση αυτή περνάει το σήμα εισόδου μέσα από ολόκληρο το δίκτυο,
- μέθοδος back propagate: η συνάρτηση αυτή σαρώνει το δίκτυο ξεκινώντας από τους νευρώνες εξόδου και αποθηκεύει τα βάρη των νευρώνων, εφαρμόζοντας τη σιγμοειδή συνάρτηση,
- μέθοδος train: η συνάρτηση αυτή εκπαιδεύει το δίκτυο, αποθηκεύοντας το error σε κάθε νευρώνα (καλώντας τη συνάρτηση mean squared error). Τρέχει για τον αντίστοιχο αριθμό από epochs που έχουμε δηλώσει νωρίτερα,
- μέθοδος gradient descent: η συνάρτηση αυτή ενημερώνει τα βάρη των νευρώνων, προσπαθώντας να μειώσει τη κλίση της εφαπτομένης,
- μέθοδος sigmoid: η συνάρτηση αυτή απλώς επιστρέφει για έναν αριθμό που δίνουμε ως όρισμα, τον αντίστοιχο αριθμό που προκύπτει από τον τύπο της σιγμοειδούς συνάρτησης (χρησιμοποιείται στο activation),
- μέθοδος sigmoid derivative: αντίστοιχα, η συνάρτηση αυτή επιστρέφει την παράγωγο της σιγμοειδούς (χρησιμοποιείται στο back propagation),
- μέθοδος mse: η συνάρτηση αυτή επιστρέφει τον μέσο όρο στο τετράγωνο, για τον αντίστοιχο αριθμό που δέχεται ως όρισμα (χρησιμοποιείται στο training)

Στη συνέχεια, όταν καλούμε τη main, δημιουργείται ένα dataset και το MLP μοντέλο με τις τιμές των παραμέτρων που θέλουμε, εκπαιδεύεται το μοντέλο, και τέλος, απλώς για παρατήρηση, δίνουμε ένα ζεύγος τιμών στο μοντέλο και βλέπουμε τι πρόβλεψη κάνει. Συγκρίνουμε την πρόβλεψη του μοντέλου με την τιμή που έχουμε δώσει ως στόχο για την έξοδο και βλέπουμε με τι ακρίβεια λειτουργεί το μοντέλο μας. Το αποτέλεσμα του τερματικού που δοκιμάσαμε εμείς φαίνεται στο αντίστοιχο παράρτημα κώδικα παρακάτω.

Στο σημείο αυτό να σημειωθεί πως ο παρακάτω κώδικας δεν είναι αποκλειστικά δικός μου. Ύστερα από αρκετή έρευνα και αναζήτηση, κατάφερα να συλλέξω αποσπάσματα κώδικα και αφού κατάλαβα τη λειτουργία κάθε συνάρτησης αλλά και του μοντέλου και της διαδικασίας ως σύνολο, κατάφερα να δημιουργήσω αυτό το πρόγραμμα σε python που φαίνεται παρακάτω.

Γραφική αναπαράσταση μετρήσεων

Figure 1: Predict time of each model

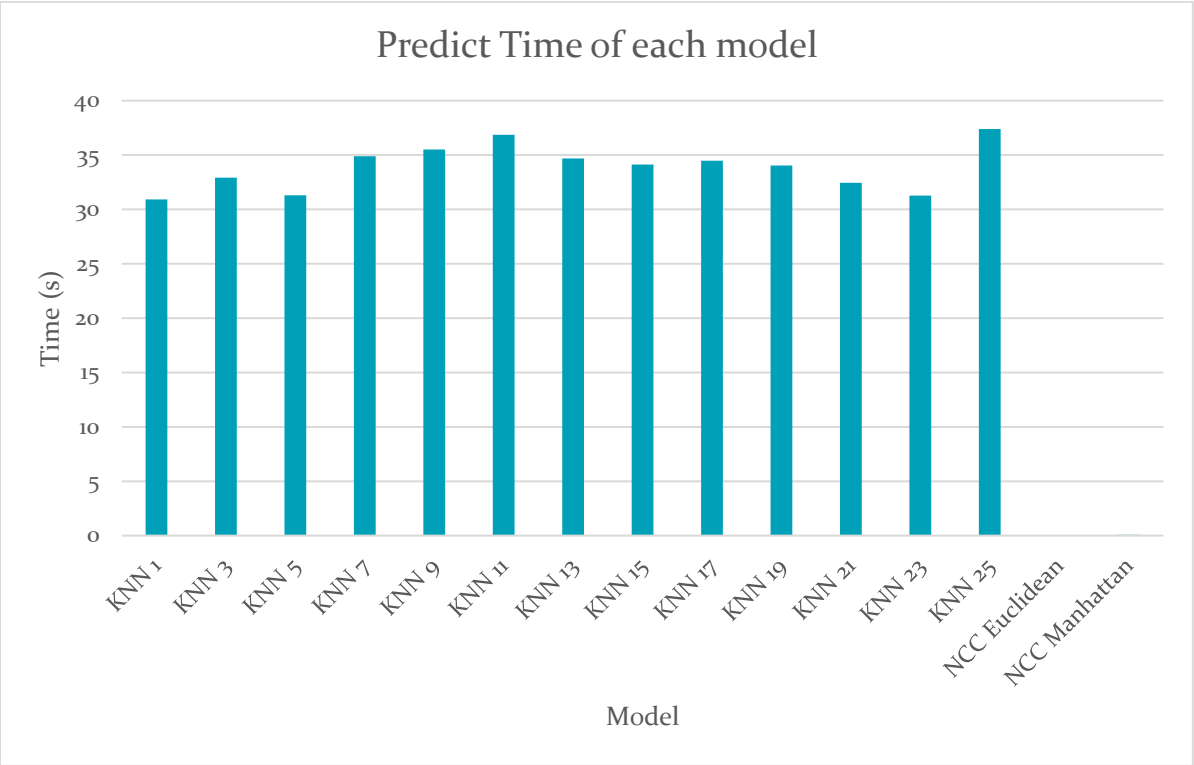


Figure 2: Fit time of each model

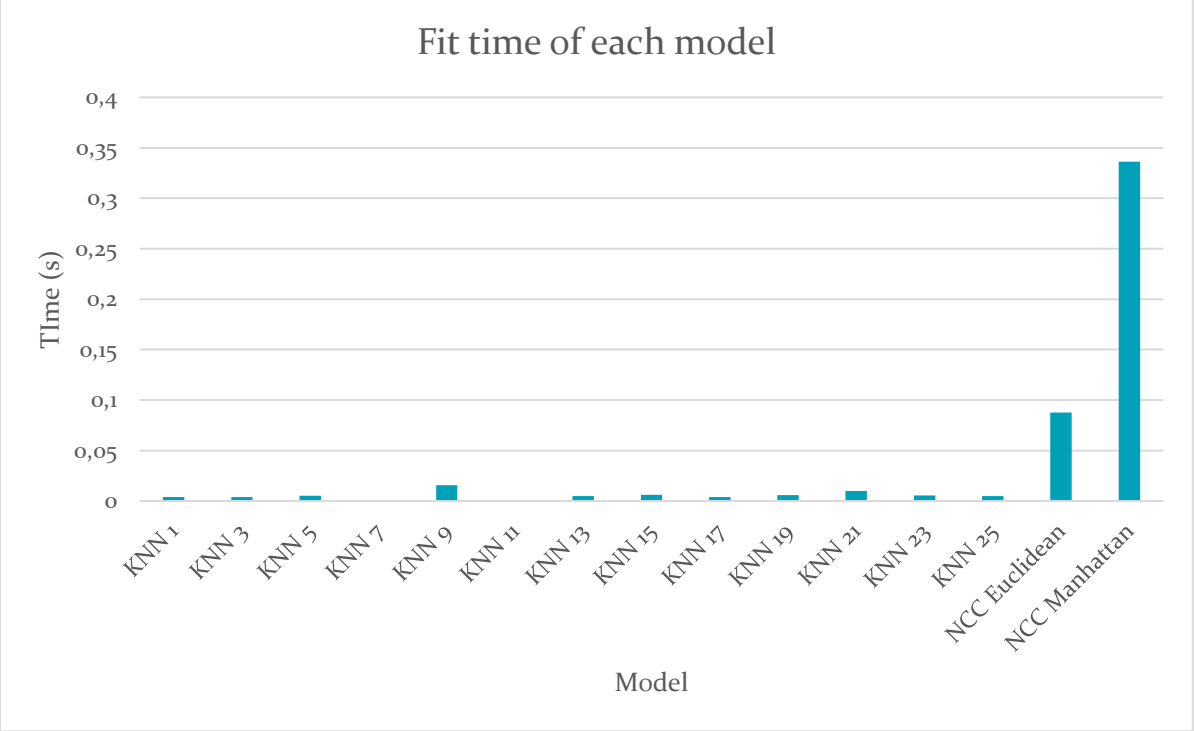


Figure 3: Total time of each model

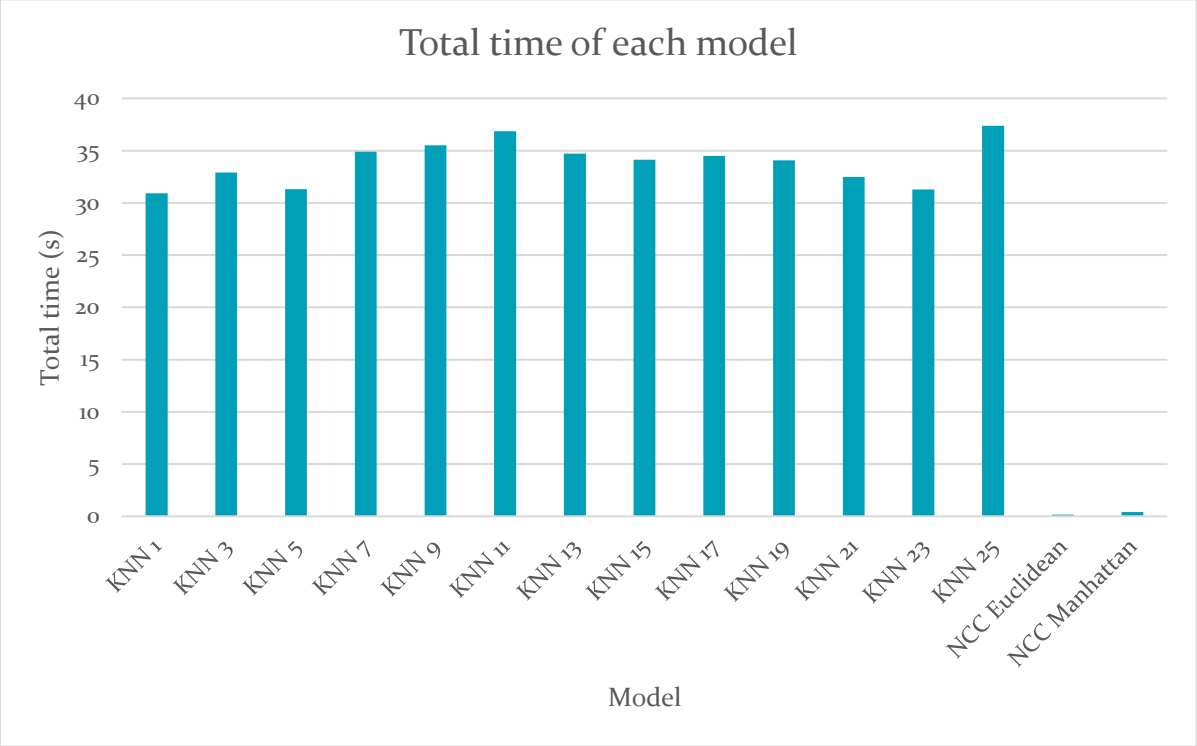


Figure 4: Accuracy percentage of KNN models

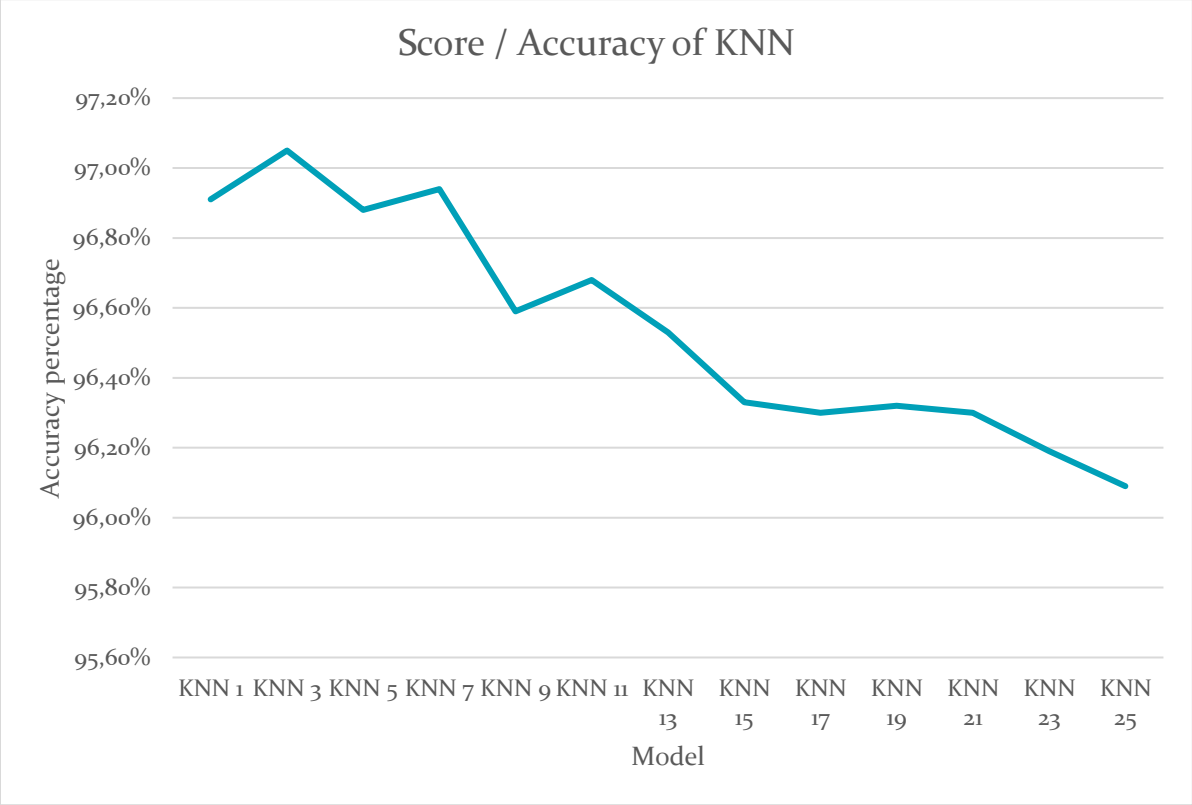


Figure 5: Accuracy percentage of each model

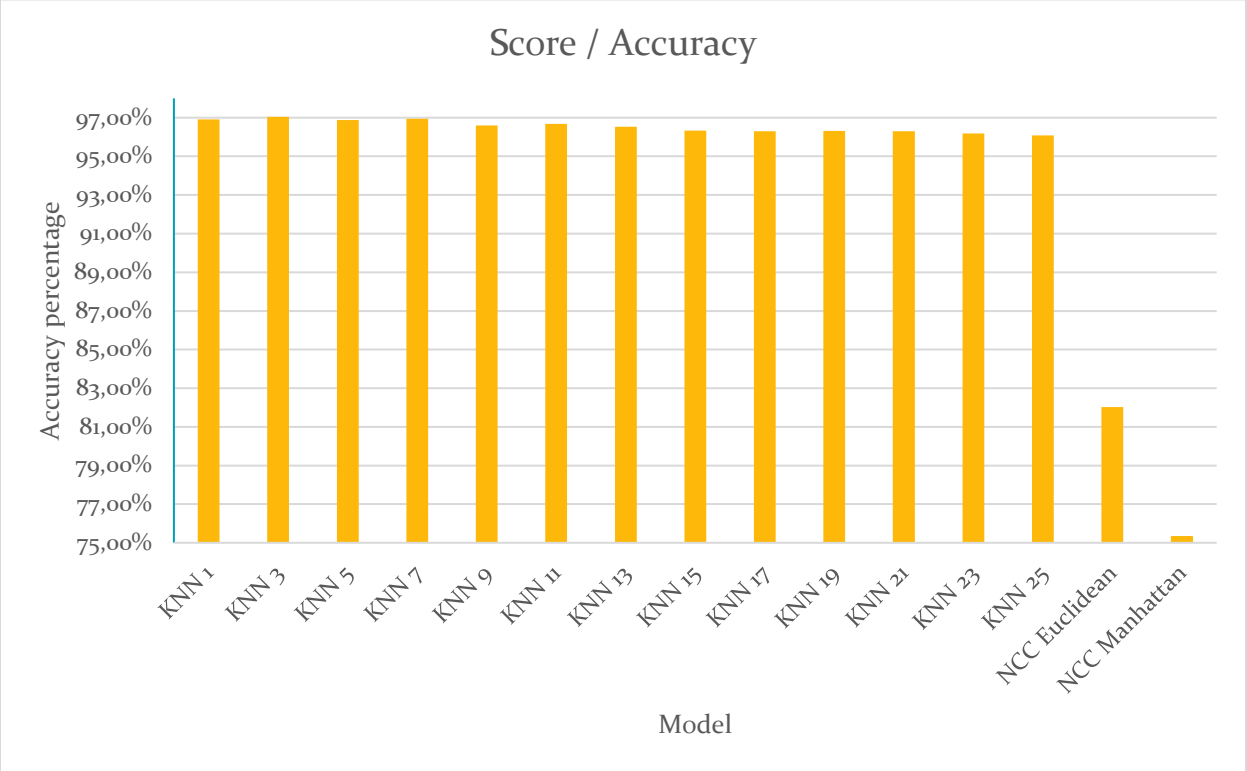


Figure 6: Accuracy percentage of KNN (3 neighbors) with Cross Validation

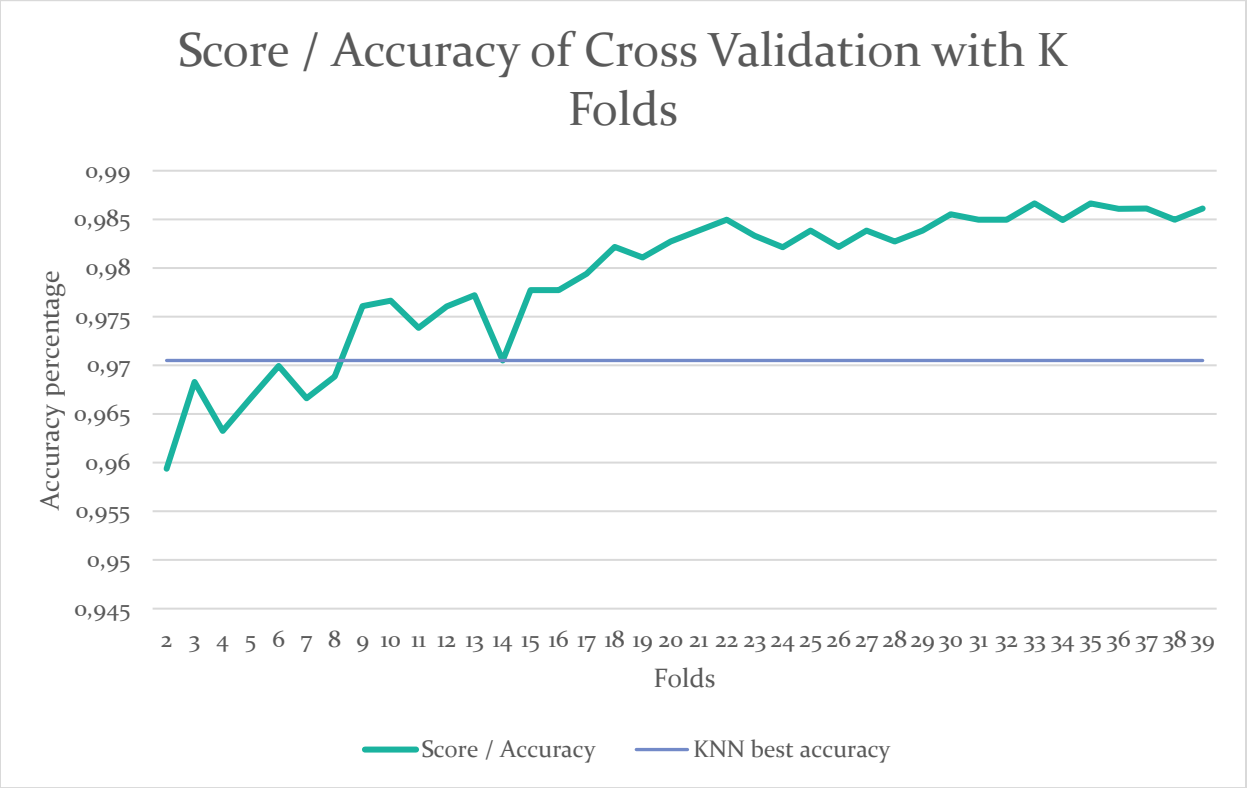


Figure 7: Accuracy of MLP per epoch vs CV Best accuracy

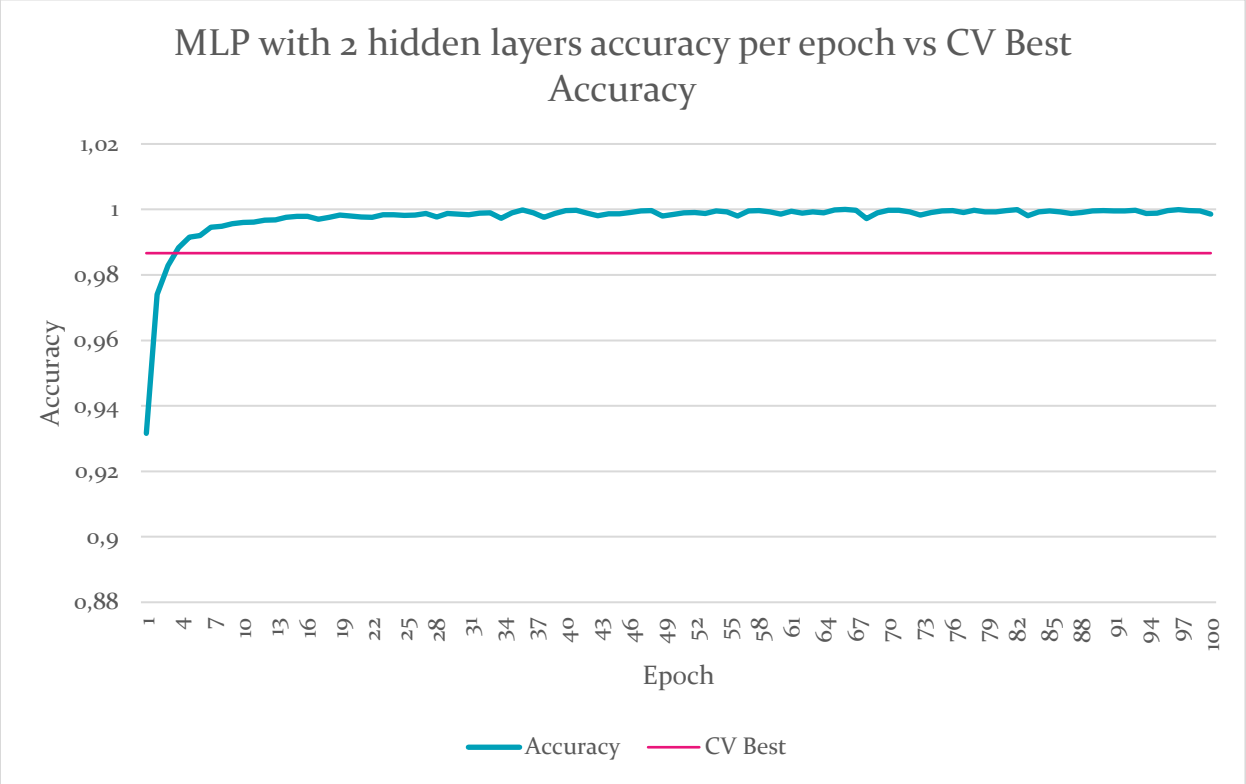


Figure 8: Loss of MLP per epoch

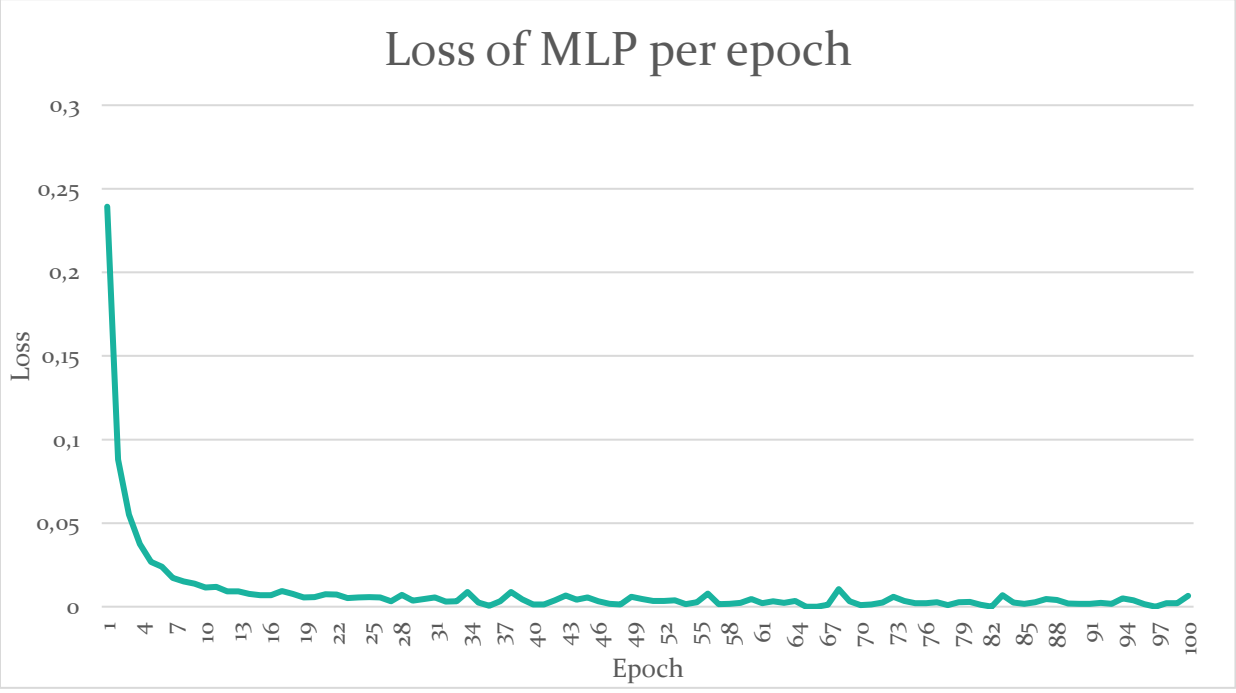


Figure 9: MLP Accuracy with 2 vs 3 hidden layers

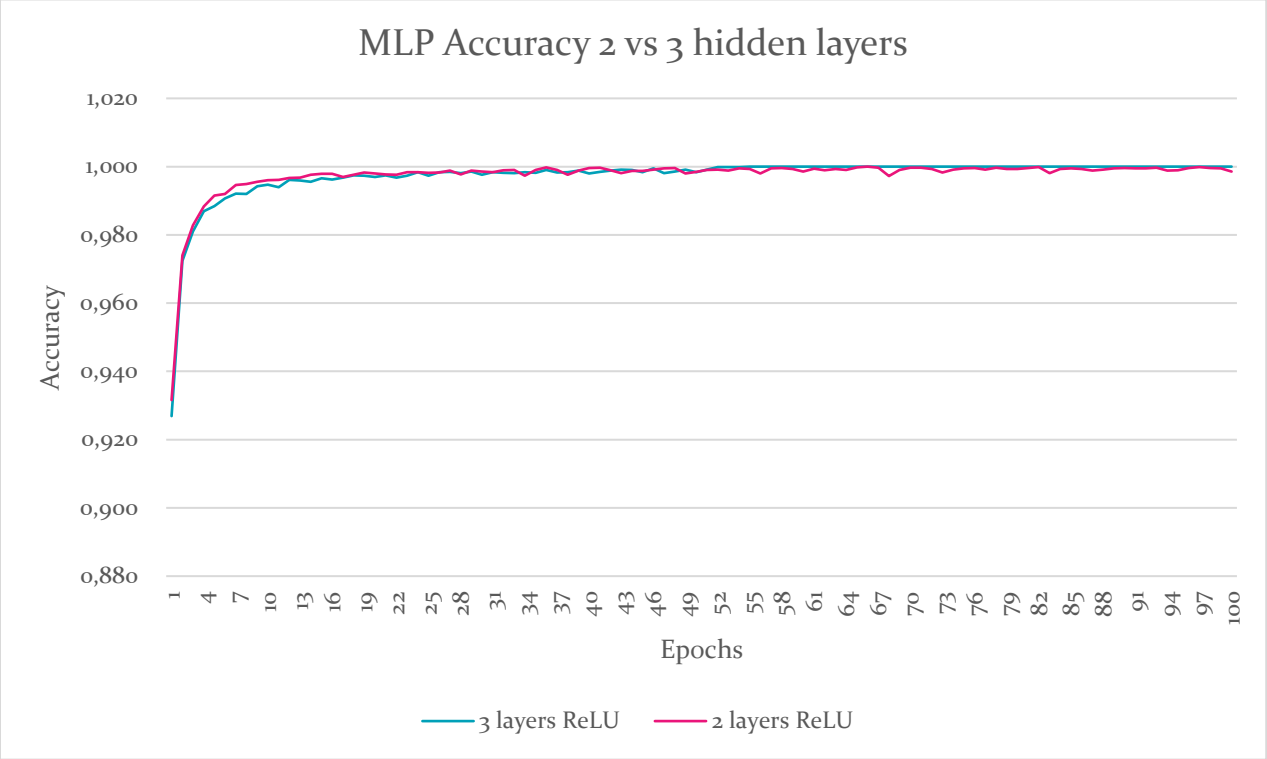


Figure 10: ReLU vs Sigmoid Activation Function Accuracy

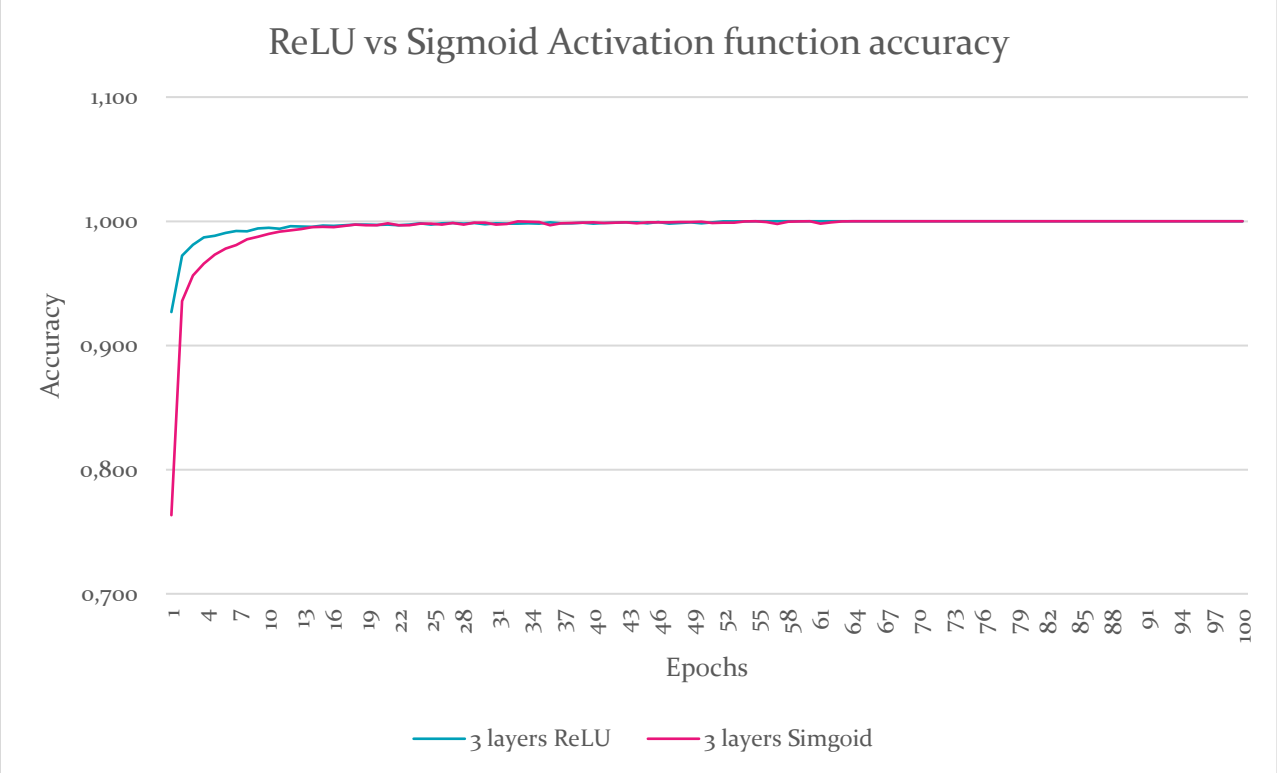
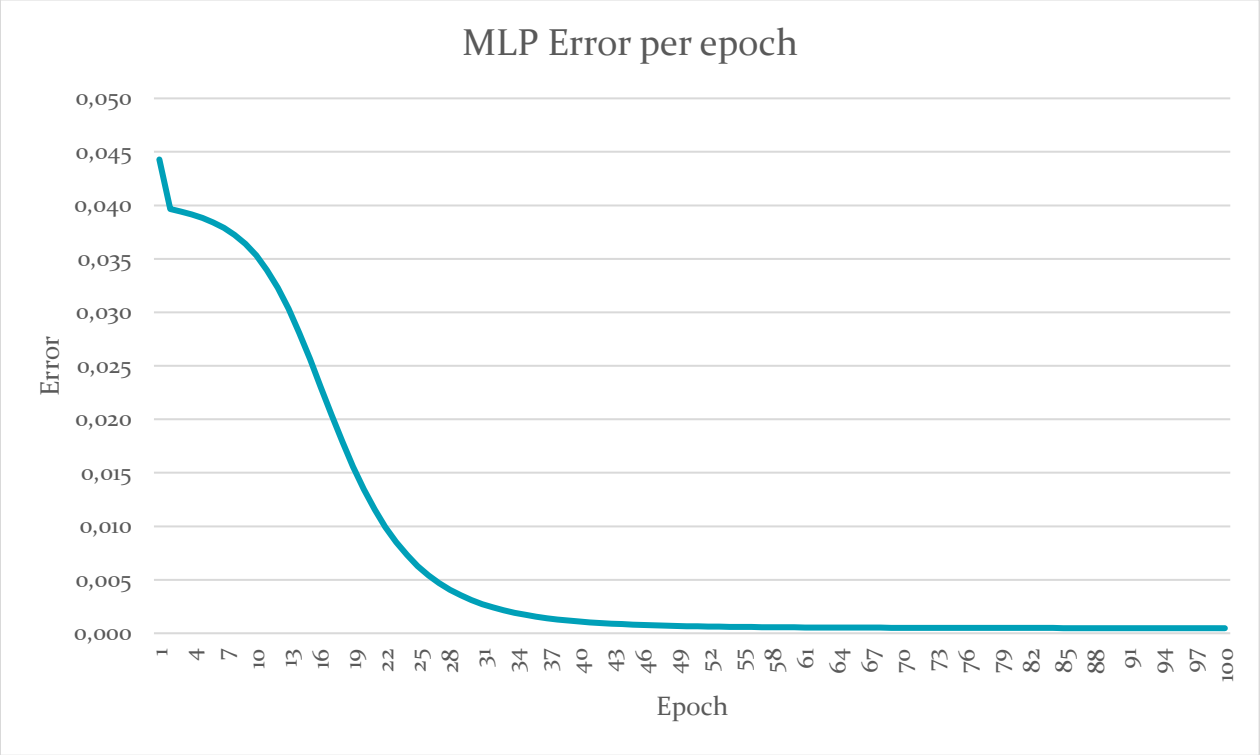


Figure 11: MLP (with back propagation) error per epoch



Σχολιασμός Αποτελεσμάτων

KNN VS NCC ACCURACY

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 ACCURACY

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.

KNN, NCC, KNN CV VS MLP ACCURACY

Παρατηρούμε από το γράφημα 7 ότι η ακρίβεια του MLP φτάνει σε εξαιρετικά υψηλές τιμές αγγίζοντας το απόλυτο 100%. Τρέχοντας το πρόγραμμα για μεγάλο αριθμό epochs παρατηρούμε πως η ακρίβεια αυξάνεται. Επίσης παρατηρούμε πως η ακρίβεια του ακόμη και για μικρό αριθμό epochs είναι υψηλότερη από την καλύτερη ακρίβεια που είχαμε μέχρι τώρα, αυτή της cross validation με ποσοστό ακρίβειας 98,66%. Επίσης ο χρόνος που τρέχει ο MLP είναι αρκετά μικρός με μέσο όρο τα 6,5 δευτερόλεπτα ανά epoch, και τα 14ms ανά step. Από το γράφημα 7 βλέπουμε ότι ακόμα και για 10 epochs έχουμε ακρίβεια κοντά στη μονάδα, σαφώς καλύτερη από αυτή της cross validation, και με μέσο χρόνο 6,5s/epoch έχουμε ακρίβεια 0,996 ή 99,6% σε χρόνο κοντά στο λεπτό (1,083 minutes). Επομένως, ο MLP ίσως χρειάζεται λίγο παραπάνω χρόνο ώστε να τρέξει για περισσότερα epochs, όμως τα ποσοστά ακρίβειάς του είναι εξαιρετικά υψηλά.

Ακόμη, παρατηρούμε από το γράφημα 9 ότι προσθέτοντας ένα ακόμα layer στον MLP, το ποσοστό της ακρίβειας αυξάνεται λιγότερο γρήγορα για περισσότερα epochs, ωστόσο φτάνουν στο 100% πιο γρήγορα και παραμένουν εκεί, για epoch > 55.

Τέλος δοκιμάσαμε την ακρίβεια του MLP με deep learning αρχιτεκτονική χρησιμοποιώντας την σιγμοειδή συνάρτηση ως συνάρτηση ενεργοποίησης, και με 3 κρυφά επίπεδα. Από τα αποτελέσματα, όπως βλέπουμε στο γράφημα 10, προκύπτει ότι παρόλο που με τη σιγμοειδή συνάρτηση ενεργοποίησης για μικρό αριθμό epochs ακρίβεια είναι κοντά στο 76%, για μεγαλύτερο αριθμό epochs μεγαλώνει η ακρίβεια φτάνοντας πάλι σε εξαιρετικά επίπεδα της τάξης του 100%.

Σχετικά με το νευρωνικό δίκτυο που υλοποιήσαμε εμείς, βλέπουμε από το γράφημα 11 και τον πίνακα 5, πως το σφάλμα (error) μειώνεται λογαριθμικά (στην αρχή απότομα και στη συνέχεια πιο ομαλά) καθώς εκπαιδεύουμε το μοντέλο με τον αλγόριθμο back propagation για μεγαλύτερο αριθμό από epochs. Και αυτό το μοντέλο είχε αρκετά μεγάλη ακρίβεια. Σαφώς θα μπορούσε να βελτιωθεί περαιτέρω με διάφορες αλλαγές, όπως αν αυξάναμε τον αριθμό από epochs, ή εάν αλλάζαμε τις συναρτήσεις ενεργοποίησης. Επίσης, να σημειωθεί πως ο χρόνος του συγκεκριμένου μοντέλου δεν μετρήθηκε καθώς το dataset που δώσαμε ως είσοδο ήταν πολύ μικρότερη της mnist, οπότε οποιαδήποτε σύγκριση δεν θα είχε νόημα.

ΓΕΝΙΚΟ ΣΥΜΠΕΡΑΣΜΑ

Τελικά, συμπεραίνουμε ότι το MLP μοντέλο, παρόλο που χρειάζεται λίγο χρόνο παραπάνω από τον KNN, τελικά πετυχαίνει εξαιρετικά επίπεδα ακρίβειας. Ο NCC είναι εξαιρετικά γρήγορος στο να κάνει predict, ωστόσο δεν έχει τόσο καλή ακρίβεια όσο άλλα μοντέλα.

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	0,41411542892456053	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: MLP Μετρήσεις Loss and Accuracy

Epoch	Loss 2 hidden layers	Accuracy 2 hidden layers, ReLU	Accuracy 3 hidden layers, ReLU	Accuracy 3 hidden layers, sigmoid
1	0,2393	0,9316	0,9269	0,763
2	0,0881	0,974	0,9724	0,936
3	0,0552	0,9828	0,9810	0,956
4	0,0375	0,9883	0,9869	0,966
5	0,0267	0,9915	0,9884	0,973
6	0,0239	0,992	0,9907	0,978
7	0,0172	0,9946	0,9921	0,981
8	0,0151	0,9949	0,9920	0,985
9	0,0137	0,9956	0,9942	0,988
10	0,0115	0,996	0,9947	0,990
11	0,0119	0,9961	0,9940	0,992
12	0,0092	0,9967	0,9961	0,993
13	0,0091	0,9968	0,9959	0,994
14	0,0076	0,9976	0,9956	0,995
15	0,0068	0,9979	0,9966	0,996
16	0,0068	0,9979	0,9962	0,995
17	0,0093	0,997	0,9968	0,996
18	0,0077	0,9976	0,9974	0,997
19	0,0055	0,9983	0,9973	0,997
20	0,0058	0,998	0,9970	0,997
21	0,0074	0,9977	0,9974	0,998
22	0,0072	0,9976	0,9968	0,997
23	0,0051	0,9984	0,9973	0,997
24	0,0055	0,9984	0,9984	0,998
25	0,0058	0,9982	0,9973	0,998
26	0,0055	0,9983	0,9984	0,997
27	0,0033	0,9988	0,9985	0,999
28	0,0071	0,9977	0,9981	0,997
29	0,0036	0,9988	0,9986	0,999

30	0,0045	0,9986	0,9976	0,999
31	0,0056	0,9984	0,9984	0,997
32	0,0031	0,9989	0,9982	0,998
33	0,0032	0,999	0,9981	1,000
34	0,0088	0,9973	0,9984	1,000
35	0,0025	0,999	0,9982	0,999
36	6,47E+08	0,9998	0,9990	0,997
37	0,0033	0,999	0,9983	0,998
38	0,0087	0,9976	0,9984	0,999
39	0,0043	0,9988	0,9988	0,999
40	0,0013	0,9996	0,9980	0,999
41	0,0014	0,9997	0,9985	0,999
42	0,0038	0,9989	0,9988	0,999
43	0,0067	0,9981	0,9991	0,999
44	0,0042	0,9987	0,9990	0,998
45	0,0055	0,9987	0,9984	0,999
46	0,0032	0,9991	0,9995	0,999
47	0,0017	0,9995	0,9981	0,999
48	0,0014	0,9996	0,9986	0,999
49	0,0059	0,998	0,9991	0,999
50	0,0045	0,9985	0,9984	1,000
51	0,0034	0,999	0,9991	0,999
52	0,0034	0,9991	0,9999	0,999
53	0,0038	0,9988	0,9999	0,999
54	0,0016	0,9995	0,9999	1,000
55	0,0027	0,9993	1,0000	1,000
56	0,0078	0,998	1,0000	0,999
57	0,0016	0,9995	1,0000	0,998
58	0,0018	0,9996	1,0000	1,000
59	0,0023	0,9993	1,0000	1,000
60	0,0046	0,9986	1,0000	1,000
61	0,0021	0,9994	1,0000	0,998

62	0,0033	0,9989	1,0000	0,999
63	0,0022	0,9993	1,0000	1,000
64	0,0035	0,999	1,0000	1,000
65	4,78E+08	0,9998	1,0000	1,000
66	5,32E+09	1	1,0000	1,000
67	0,0012	0,9997	1,0000	1,000
68	0,0106	0,9972	1,0000	1,000
69	0,0032	0,999	1,0000	1,000
70	8,90E+08	0,9997	1,0000	1,000
71	0,0013	0,9997	1,0000	1,000
72	0,0025	0,9993	1,0000	1,000
73	0,0059	0,9983	1,0000	1,000
74	0,0034	0,9991	1,0000	1,000
75	0,002	0,9995	1,0000	1,000
76	0,0021	0,9996	1,0000	1,000
77	0,0026	0,9991	1,0000	1,000
78	0,001	0,9997	1,0000	1,000
79	0,0027	0,9993	1,0000	1,000
80	0,0028	0,9993	1,0000	1,000
81	0,0012	0,9996	1,0000	1,000
82	2,90E+08	0,9999	1,0000	1,000
83	0,0069	0,9981	1,0000	1,000
84	0,0025	0,9993	1,0000	1,000
85	0,0017	0,9995	1,0000	1,000
86	0,0026	0,9993	1,0000	1,000
87	0,0046	0,9988	1,0000	1,000
88	0,004	0,9991	1,0000	1,000
89	0,0019	0,9995	1,0000	1,000
90	0,0017	0,9996	1,0000	1,000
91	0,0017	0,9995	1,0000	1,000
92	0,0022	0,9995	1,0000	1,000
93	0,0018	0,9997	1,0000	1,000

94	0,0049	0,9988	1,0000	1,000
95	0,0038	0,9989	1,0000	1,000
96	0,0016	0,9996	1,0000	1,000
97	4,58E+08	0,9999	1,0000	1,000
98	0,002	0,9996	1,0000	1,000
99	0,0021	0,9995	1,0000	1,000
100	0,0064	0,9986	1,0000	1,000

Table 5: MLP (with back propagation) error per epoch

Epoch	Error
1	0.044287450680812 58
10	0.035312851337593563
20	0.013450604798935745
30	0.003105254693212138
40	0.0010992727338264001
50	0.0006645159411769947
60	0.0005523049273852001
70	0.0005148518972380941
80	0.0004963912756482527
90	0.0004834460643046681
100	0.0004726237601748367

(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)
```

MLP (Deep learning architecture)

```python
# import the libraries
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense

# load the dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# reshape the dataset
x_train = x_train.reshape(60000, 784).astype('float32')
x_test = x_test.reshape(10000, 784).astype('float32')

# normalize the dataset
x_train = x_train / 255.0
x_test - x_test / 255.0

y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

# choose model properties like hidden layers, activation functions and number of neurons
model = Sequential()
model.add(Dense(512, input_shape=(784,), activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))

# compile
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['acc'])

# fit
model.fit(x_train, y_train, epochs=100, batch_size=128, validation_data=(x_test, y_test))
```

MLP (with back propagation)

```python
# import the libraries
import numpy as np
from random import random


# MLP Class
class MLP(object):

    # Here we initialize the MLP, giving the numbers of inputs, the number of hidden
    # layers and the number of outputs
    def __init__(self, num_inputs=3, hidden_layers=[3, 3], num_outputs=2):

        self.num_inputs = num_inputs
        self.hidden_layers = hidden_layers
        self.num_outputs = num_outputs

        # create general representation of layers
        layers = [num_inputs] + hidden_layers + [num_outputs]

        # create random weights of connections between layers
        weights = []
        for i in range(len(layers) - 1):
            w = np.random.rand(layers[i], layers[i + 1])
            weights.append(w)
        self.weights = weights

        # store the derivatives of each layer (we need these for the back propagation algorithm)
        derivatives = []
        for i in range(len(layers) - 1):
            d = np.zeros((layers[i], layers[i + 1]))
            derivatives.append(d)
        self.derivatives = derivatives

        # save activations per layer
        activations = []
        for i in range(len(layers)):
            a = np.zeros(layers[i])
            activations.append(a)
        self.activations = activations
```

```python
    # function to forward propagate the MLP
    def forward_propagate(self, inputs):

        # the input layer activation is just the input itself
        activations = inputs

        # store the activations for back propagation algorithm
        self.activations[0] = activations

        # iterate through the network layers
        for i, w in enumerate(self.weights):
            # calculate matrix multiplication between previous activation and weight matrix
            net_inputs = np.dot(activations, w)

            # apply sigmoid activation function
            activations = self._sigmoid(net_inputs)

            # store activations for back propagation algorithm
            self.activations[i + 1] = activations

        return activations

    # back propagation algorithm function
    def back_propagate(self, error):

        # iterate backwards through the network layers
        for i in reversed(range(len(self.derivatives))):
            # get activation of previous layer
            activations = self.activations[i + 1]

            # apply sigmoid derivative function
            delta = error * self._sigmoid_derivative(activations)

            # reshape into 2d
            delta_re = delta.reshape(delta.shape[0], -1).T

            # get activations for current layer
            current_activations = self.activations[i]

            # reshape activations to 2d matrix
            current_activations = current_activations.reshape(current_activations.shape[0], -1)

            # store derivative after matrix multiplication
            self.derivatives[i] = np.dot(current_activations, delta_re)

            # back propagate the next error
            error = np.dot(delta, self.weights[i].T)
```

```
    # function to train the model
    def train(self, inputs, targets, epochs, learning_rate):

        # now enter the training loop
        for i in range(epochs):
            sum_errors = 0

            # iterate through all the training data
            for j, input in enumerate(inputs):
                target = targets[j]

                # activate the network!
                output = self.forward_propagate(input)

                error = target - output

                self.back_propagate(error)

                # update the weight by applying gradient descent on derivatives
                self.gradient_descent(learning_rate)

                sum_errors += self._mse(target, output)

            # print error per epoch
            print("Error: {} at epoch {}".format(sum_errors / len(items), i + 1))

        print("Training done")

    # descends the gradient to help the training process
    def gradient_descent(self, learningRate=1):

        # update the weights by stepping down the gradient
        for i in range(len(self.weights)):
            weights = self.weights[i]
            derivatives = self.derivatives[i]
            weights += derivatives * learningRate

    # sigmoid function used for activation
    def _sigmoid(self, x):

        y = 1.0 / (1 + np.exp(-x))
        return y

    # sigmoid derivative function used for error
    def _sigmoid_derivative(self, x):

        return x * (1.0 - x)

    # mse function (mean squared error)
    def _mse(self, target, output):

        return np.average((target - output) ** 2)
```

```
if __name__ == "__main__":
    # create a dataset to train a network for the sum operation
    items = np.array([[random() / 2 for _ in range(2)] for _ in range(1000)])
    targets = np.array([[i[0] + i[1]] for i in items])

    # create a Multilayer Perceptron with one hidden layer
    mlp = MLP(2, [5], 1)

    # train the model
    mlp.train(items, targets, 50, 0.1)

    # testing set
    input = np.array([0.3, 0.1])
    target = np.array([0.4])

    # predict
    output = mlp.forward_propagate(input)

    # testing the model
    print()
    print("{} + {} is equals {}".format(input[0], input[1], output[0]))
```

```
0.4 + 0.3 is equals 0.707876362921217

Process finished with exit code 0
```