Νευρωνικά Δίκτυα - Βαθιά Μάθηση

Πρώτη Υποχρεωτική Εργασία

Ευαγγελία Δημαρίδου ΑΕΜ: 10410

I. Εισαγωγή

Το θέμα της παρούσας εργασίας είναι να υλοποιήσουμε ένα νευρωνικό δίκτυο πολυστρωματικού perceptron που θα χρησιμοποιεί για την εκπαίδευση του τον αλγόριθμο back propagation. Το δίκτυο αυτό χρησιμοποιείται για την κατηγοριοποίηση δεδομένων, και στη συγκεκριμένη εργασία για το σύνολο δεδομένων της Cifar-10. Αυτή περιέχει εικόνες οι οποίες ανήκουν σε 10 κλάσεις. Ακολούθηθηκε υλοποίηση σε python, αρχικά με χρήση της numpy χωρίς έτοιμες βιβλιοθήκες, αλλά και σε pytorch. Μάλιστα στην pytorch υλοποιήθηκε και νευρωνικό δίκτυο με convolutional layers.

II. Υλοποίηση FROM SCRATCH

A. Φόρτωση Δεδομένων

Τα δεδομένα εισόδου της Cifar-10 αποτελούνται από 6 batches με 10000 δείγματα το καθένα. Στην συγκεκριμένη υλοποίηση, τα πρώτα 4 χρησιμοποιούνται ως training data το 5ο ως validation data και το τελευταίο ως test data. Γίνεται αποσυμπίεση τους χρησιμοποιώντας την συνάρτηση unpickle() που αναφέρεται στην ιστοσελίδα της Cifar-10. Τα δείγματα χωρίζονται σε data και labels και στην συνέχεια τα trainining data ενώνονται σε έναν numpy πίνακα με την συνάρτηση np.concatenate().

B. Επεξεργασία Δεδομένων

- Καθώς το κάθε στοιχείο των δεδομένων έχει τύπο uint8 δηλαδή αποτελείται από 8 bits, διαιρούμε με το 255 προκειμένου να έχουμε numerical stability.

- Όσον αφορά τις κλάσεις (labels) των δεδομένων, μετατρέπονται σε one hot encoded πίνακες χρησιμοποιώντας την συνάρτηση one_hot_encoding(), η οποία επιστρέφει numpy πίνακα μεγέθους (data_size x 10). Για το training dataset για παράδειγμα επιστρέφει πίνακα 40000 x 10. Για ένα δείγμα, ο πίνακας έχει τιμή 1 μόνο στο index της κλάσης στην οποία ανήκει.

C. Οργάνωση Κώδικα

Για την υλοποίηση του νευρωνικού δικτύου from scratch, δημιουργήθηκαν δύο κλάσεις.

1) Layer

Η πρώτη είναι η κλάση layer που αντιπροσωπεύει ένα στρώμα του perceptron.

- Οι μεταβλητές οι οποίες έχει είναι το μέγεθος εισόδου (input_size), το μέγεθος του επόμενου layer (output_size), τα βάρη καθώς και το bias. Τα βάρη (weights) και το bias είναι μεταβλητές του layer που προηγείται. Δηλαδή σε ένα δίκτυο με μόνο 2 hidden layers, τα βάρη και τα biases ανήκουν στην κλάση του πρώτου layer.

- Για την αρχικοποίηση των βαρών με την συνάρτηση initialize_weights() χρησιμοποιήθηκαν τεχνικές που λειτουργούν αποδοτικά μαζί με την συνάρτηση ενεργοποίησης. Καθώς πρόκειται για πρόβλημα κατηγοριοποίησης, χρησιμοποιήθηκε η συνάρτηση relu και έτσι τα βάρη αρχικοποιούνται με he initiliazation. Τα βάρη είναι ένας numpy πίνακας 2 διαστάσεων, (output_size x input_size). Τα biases ειναι επίσης numpy arrays, μονοδιάστατα με μέγεθος την διάσταση του επόμενου layer και αρχικοποιούνται με το 0.

2) neural_network

Η δεύτερη κλάση είναι η neural_network. Είναι ουσιαστικά η κλάση που πραγματοποιεί την διαδικασία εκπαίδευσης και επαλήθευσης.

- Η συνάρτηση αρχικοποίησης, "__init__" δημιουργεί όλα τα απαιτούμενα layers που της έχουν δοθεί. Συγκεκριμένα δίνεται μία λίστα με τα hidden layers, το μέγεθος εισόδου καθώς και εξόδου. Τα layers είναι αντικείμενα της κλάσης layer και δημιουργούνται ανάλογα με τα αντίστοιχα μεγέθη. Σε αυτή τη συνάρτηση δηλώνονται και 4 numpy πίνακες στους οποίους αποθηκεύονται τα training accuracy and loss καθώς και validation accuracy και loss.

- Η συνάρτηση feed_forward() περνάει ένα δείγμα από όλα τα layers του νευρωνικού δικτύου. Ορίζονται 2 λίστες post_activation και pre_activation στις οποίες αποθηκεύονται οι αντίστοιχες τιμές. Στο πρώτο pre_activation αποθηκεύεται το εσωτερικό γινόμενο των βαρών με το sample αυξημένο κατά το bias του input layer. Το αντίστοιχο pre_activation numpy array προστίθεται στην λίστα. Το post_activation είναι ουσιαστικά το pre_activation στο οποίο εφαρμόζεται η συνάρτηση ενεργοποίησης. Στην συγκεκριμένη υλοποίηση επιλέγεται η relu. Στα υπόλοιπα εσωτερικά layers πραγματοποιείται η ίδια διαδικασία και στο τελευταίο εφαρμόζεται η softmax. Με την softmax, οι 10 τιμές του output layer μετατρέπονται σε 10 θετικούς αριθμούς με άθροισμα 1 και έτσι τις χρησιμοποιούμε ως πιθανότητες. Προκειμένου οι τιμές αυτές να αξιοποιηθούν και από την συνάρτηση

back_propagation(), αποθηκεύονται ως μεταβλητές της κλάσης στο τέλος της συνάρτησης.

- Η συνάρτηση compute_error() παίρνει ως παράμετρο το target output, το οποίο είναι ενα one hot encoded numpy array, δηλαδή έχει τιμή 1 μόνο στην κλάση στην οποία ανήκει το δείγμα που εξετάζεται. Έτσι πραγματοποιείται αφαίρεση μεταξύ του target και του self.post_activation του τελευταίου layer.

- Η συνάρτηση back_propagation() υλοποιεί τον αλγόριθμο που δηλώνει το όνομα της. Εφαρμόζει τις γνωστές εξισώσεις δημιουργώντας αρχικά μία λίστα delta_list. Το δέλτα του εξωτερικού layer είναι το self.error_o που υπολογίστηκε στην συνάρτηση compute_error(), και τα υπόλοιπα υπολογίζονται από την σχέση:

$$\delta_j = \varphi'_j(u_j) * \sum_{\kappa} \delta_\kappa * w_{\kappa_j}$$

Το κάθε layer έχει το δικό του δέλτα και από τις ιδιότητες της numpy απαλείφουμε τους δείκτες και ουσιαστικά πραγματοποιούμε εσωτερικό γινόμενο μεταξύ του δέλτα και του W, κάνοντας τους απαραίτητους μετασχηματισμούς μεγέθους. Έπειτα αναστρέφουμε τη λίστα με τα δέλτα έτσι ώστε το πρώτο στοιχείο να αναφέρεται στο δέλτα του πρώτου hidden layer.

- Η συνάρτηση gradient_descent() παίρνει ως ορίσματα το συγκεκριμένο δείγμα εκπαίδευσης με το οποίο ασχολούμαστε και το learning rate και εφαρμόζει τις εξισώσεις:

$$\Delta w_{j_i} = eta * \delta_j * y_i$$

$$\Delta \beta_j = eta * \delta_j$$

Καθώς τα βάρη αποθηκεύονται σε έναν 2d array το παραπάνω μπορεί να πραγματοποιηθεί με ένα εσωτερικό γινόμενο του δέλτα και του post_activation του προηγούμενου layer. Το μέγεθος του δέλτα θα είναι (current_layer x 1), και της εισόδου (1 x layer_before) άρα συνολικά παίρνουμε το επιθυμητό μέγεθος. Για τα βάρη μεταξύ input_layer και πρώτου hidden_layer η εξίσωση διαφοροποιείται, αφού χρησιμοποιείται το train_data καθώς αυτή είναι η είσοδος του.

- Η συνάρτηση train_network() χρησιμοποιεί τις παραπάνω συναρτήσεις προκειμένου να εκπαιδεύσει το νευρωνικό δίκτυο. Οι παράμετροι της είναι τα δεδομένα εκπαίδευσης (train_data) και οι κλάσεις τους (train_labels, τα δεδομένα επαλήθευσης (val_data) και οι κλάσεις τους (val_labels), ο αριθμός των εποχών (epoch_count/) και το learning rate. Τα train_data είναι numpy πίνακας με μέγεθος training_size * sample_size δηλαδή 40000 x 3072. Το σύνολο επαλήθευσης έχει μέγεθος 10000 x 3072. Προκειμένου τα δείγματα να περνάνε σε κάθε εποχή με διαφορετική σειρά από την διαδικασία εκπαίδευσης, πραγματοποιείται ανακατανομή μεταξύ τους με την χρήση της np.random.permutation(). Έτσι για κάθε εποχή το κάθε δείγμα εκπαίδευσης περνάει από την feed_forward(), compute_error_o(), back_propagation() και gradient_descent() αντίστοιχα.

- Στην συνάρτηση train_network() υπολογίζονται και τα train_accuracy, train_loss, χρησιμοποιώντας cross entropy loss που είναι ο ενδεδειγμένος τρόπος υπολογισμού του loss όταν χρησιμοποιείται η softmax ως συνάρτηση ενεργοποίησης του εξωτερικού layer. Αυτό υπολογίζεται για το ένα δείγμα ως εξής:

$$loss = \sum_{\kappa=1}^{size} target\_probability * log(output\_probability)$$

Το μέγεθος είναι 10 στην περίπτωση μας αφού υπάρχουν 10 κλάσεις. Έτσι ο μέσος όρος του loss, αφού υπολογιστεί για κάθε δείγμα εκπαίδευσης, αποτελεί το training loss της εποχής αυτής.
Το accuracy υπολογίζεται από την σύγκριση της κλάσης με την μεγαλύτερη πιθανότητα στο post_activation του τελευταίου layer με την πραγματική κλάση που ανήκει το δείγμα. Το πρώτο υπολογίζεται με την συνάρτηση np.argmax() που επιστρέφει το index του στοιχείου με τη μεγαλύτερη τιμή. Έτσι υπολογίζεται το ποσοστό επί τοις εκατό το οποίο αποθηκεύεται στην αντίστοιχη λίστα train_accuracy.

- Η συνάρτηση test_network() χρησιμοποιείται για την επαλήθευση του νευρωνικού δικτύου με δείγματα στα οποία δεν έχει εκπαιδευθεί να αναγνωρίζει. Δέχεται ως παραμέτρους τα δείγματα και τις κλάσεις τους και το αν πρόκειται για test ή validation set. Περνάει το κάθε δείγμα από την συνάρτηση feed_forward() και υπολογίζει validation accuracy και loss αν πρόκειται για validation, και αλλιώς εμφανίζει το testing accuracy.

- Στην διάρκεια των πειραμάτων χρειζόμαστε διαγράμματα τα οποία θα δείχνουν την σχέση μεταξύ training και validation accuracy καθώς και loss, όσο περνάνε οι εποχές. Για τον σκοπό αυτό δημιουργούμε την συνάρτηση plot_accur_loss() η οποία είναι μέρος της κλάσης neural_network. Χρησιμοποιούμε συναρτήσεις του matplotlib.pyplot καθώς και τα περιεχόμενα από τις λίστες των loss και accuracy που είναι μεταβλητές της κλάσης neural_network.

III. PYTORCH DENSE LAYERS

Για την εργασία αυτή αξιοποιήθηκε και η βιβλιοθήκη της pytorch, προκειμένου να υπάρξει εξοικείωση καθώς και για να συγκριθεί με την from scratch υλοποίηση.

A. Δεδομένα Εισόδου

Εδώ τα δεδομένα φορτώνονται μέσω της torchvision.datasets.CIFAR10() και χωρίζονται τυχαία κατά 80% σε training και 20% σε validation. Αυτό επιτυγχάνεται μέσω της συνάρτησης torch.utils.data.random_split(). Ακόμη πραγματοποιείται επεξεργασία των δεδομένων και συγκεκριμένα

αξιοποιώντας τη βιβλιοθήκη transforms, ορίζεται transform που μετατρέπει την είσοδο σε tensor με τιμές από 0 έως 1 και στη συνέχεια το κανονικοποιεί μεταξύ -1 και 1. Χρησιμοποιήθηκε καθώς είναι μία κοινή πρακτική στα νευρωνικά δίκτυα.

B. neural_network

Για την υλοποίηση του νευρωνικού χρησιμοποιήθηκε παρόμοια λογική με την from scratch υλοποίηση. Δημιουργήθηκε μία κλάση neural_network η οποία αυτή τη φορά κληρονομεί την nn.Module και έτσι μπορεί να αξιοποιήσει την pytorch.

- Τα layers αυτή τη φορά είναι αντικείμενα του nn.Linear και χρησιμοποιούμε kaiming_uniform initiliazation τόσο για τα βάρη όσο και τα biases. Είναι ουσιαστικά he initialization που συνδυάζεται με την relu ως συνάρτηση ενεργοποίησης. Τα layers ορίζονται πάλι από τον χρήστη και αποθηκεύονται σε nn.ModuleList() προκειμένου να αναγνωρίζονται οι ιδιότητες τους απο την pytorch.

- Η συνάρτηση forward παίρνει ως όρισμα το κάθε δείγμα και το περνάει από όλα τα layers στην λίστα καθώς και από το input layer που ορίζεται ξεχωριστά. Στο τελευταίο layer δεν εφαρμόζεται η relu καθώς η pytorch εφαρμόζει αυτόματα την softmax όταν οριστεί η χρήση nn.CrossEntropyLoss().

- Η συνάρτηση train_network πραγματοποιεί την διαδικασία εκπαίδευσης και σε αυτή ορίζεται ο optimizer καθώς και το loss που χρησιμοποιείται. Της δίνονται ως ορίσματα τα train_dataloader, train_size, val_dataloader, val_size, epochs που είναι ο αριθμός των εποχών και eta το learning rate.
  Αρχικά για κάθε δείγμα εκπαίδευσης, τα gradients του optimizer ορίζονται ως 0 με την συνάρτηση self.gradient_descne.zero_grad(). Έπειτα καλώντας τη συνάρτηση my_neural_network(train_data) με όρισμα το δείγμα, καλείται η συνάρτηση forward. Υπολογίζεται ακολούθως το loss με την αντίστοιχη συνάρτηση self.cr_entr_loss() και έπειτα πραγματοποιείται back propagation και gradient descent μέσω των backward() και gradient_descent.step().

- Το accuracy υπολογίζεται μέσω της σύγκρισης της κλάσης στην οποία ανήκει το δείγμα με αυτή που υπολογίζει το νευρωνικό δίκτυο. Χρησιμοποιείται η torch.max(), παίρνοντας το δεύτερο output της που είναι το index της μεγαλύτερης πιθανότητας και άρα της εκτιμώμενης κλάσης.

- Η συνάρτηση test_network() ακολουθεί την φιλοσοφία της pytorch και έτσι περνάει το κάθε δείγμα (test ή validation, ανάλογα με το keyword), από την συνάρτηση forward έχοντας πρώτα ορίσει with torch.no_grad() εφόσον θα πραγματοποιηθεί μόνο feed_forward.
  Το accuracy και το loss υπολογίζονται με τον ίδιο τρόπο όπως και πριν.

- Διαγράμματα που απεικονίζουν τα training και validation loss και accuracy δημιουργούνται με την συνάρτηση plot_accur_loss με παρόμοια λογική όπως και στην from scratch υλοποίηση.

IV. Convolutional Neural Network

Καθώς παρατηρήθηκε ότι ακόμα και με αρκετές τροποποιήσεις των δικτύων, το validation accuracy δεν ξεπέρασε το 50 % και εφόσον αντιμετωπίζουμε πρόβλημα κατηγοριοποίησης εικόνων, δημιουργήθηκε ένα CNN με τη βοήθεια της Pytorch.

A. Κώδικας *CNN*

Προκειμένου να δημιουργηθεί το δίκτυο αυτό, χρησιμοποιήθηκε ο κώδικας των dense layers ως βάση.

1) Αλλαγές

- Τα convolutional layers ουσιαστικά αποτελούν φίλτρα με μέγεθος που ορίζεται από τον χρήστη. Χρησιμοποιούνται αρκετά για εικόνες οι οποίες είναι δύο διαστάσεων και ουσιαστικά το φίλτρο πραγματοποιεί ένα πολλαπλασιασμό element-wise με τα pixels της εικόνας. Δηλαδή σε αυτή την περίπτωση τα δεδομένα για κάθε δείγμα δεν πρέπει να είναι σε ένα κανάλι, αλλά θα πρέπει να είναι χωρισμένα σε 3 κανάλια καθώς είναι RGB. Το καθένα μάλιστα θα έχει μέγεθος 32 x 32. Αυτό έχεις ως αποτέλεσμα να χρειάζονται δύο αλλαγές στην επεξεργασία των δεδομένων. Αρχικά πλέον το flattening των δεδομένων στην συνάρτηση train_network() πρέπει να αφαιρεθεί, αφού τα δεδομένα πρέπει στην αρχή της συνάρτησης forward να έχουν 3 κανάλια. Η άλλη αλλαγή είναι ότι ανάλογα με τα convolutional layers που έχουν χρησιμοποιηθεί, το μέγεθος των δεδομένων που περνάνε μέσα από τα dense layers προσαρμόζεται ανάλογα.

- Στις μεταβλητές της κλάσης neural_network, η οποία τώρα μετονομάστηκε σε conv_neural_network, προστίθενται και τα Convolutional και MaxPool layers.

V. Πειράματα

Για το επόμενο μέρος της εργασίας πραγματοποιήθηκαν μία σειρά από πειράματα με διαφορετικές παραμέτρους.

A. Πείραμα *1*

- Το πείραμα αυτό αφορά την from scratch υλοποίηση. Αρχικά χρησιμοποιήθηκαν 3 layers με μεγέθη [110, 70, 70]. Η επιλογή έγινε μετά απο δοκιμές σε σχέση με το ποιος συνδυασμός φαίνεται να έβγαζε υψηλότερο validation accuracy στις πρώτες εποχές.
- Για αυτά τα layers χρησιμοποιήθηκε learning rate 10^{-3}

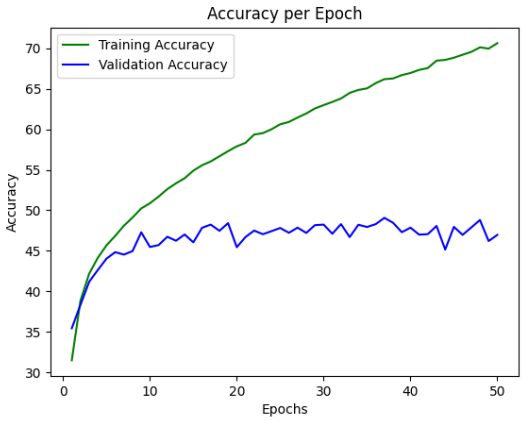


Figure 1: From Scratch, Testing Accuracy: 47.38 %

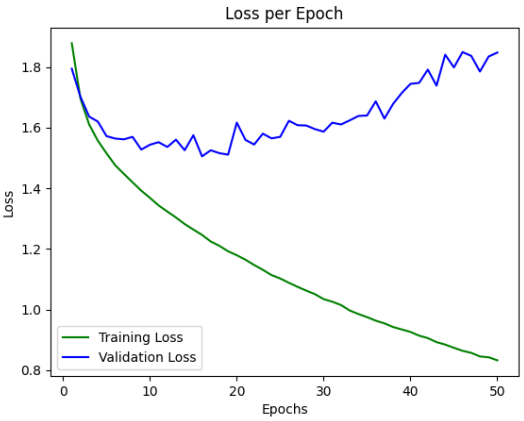


Figure 2: From Scratch

- Παρατηρούμε ότι το validation loss αυξάνεται ενώ το validation accuracy παραμένει κατά μέσο όρο σταθερό και το training accuracy ανεβαίνει, άρα υπάρχει overfitting. Αυτό μπορεί να συμβεί αν μειώνεται η πιθανότητα της σωστής κλάσης αλλά εξακολουθεί να είναι μεγαλύτερη από τις υπόλοιπες. Τέλος ο χρόνος εκπαίδευσης του πειράματος ήταν 1 ώρα και 28 λεπτά.

B. Πείραμα *2*

Πραγματοποιούμε το ίδιο πείραμα αλλά στην Pytorch χρησιμοποιώντας stochastic gradient descent ως optimizer καθώς δεν μεταβάλλει το learning rate και μπορεί να γίνει σωστή σύγκριση.

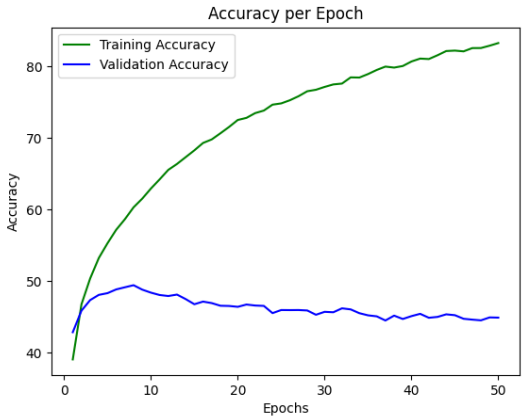


Figure 3: Pytorch, Testing Accuracy: 45.43 %

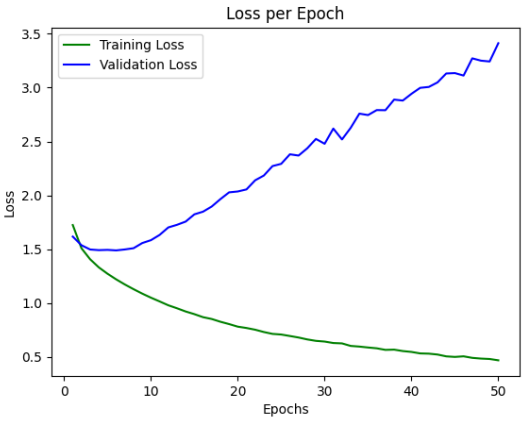


Figure 4: Pytorch

Από τα παραπάνω διαγράμματα παρατηρούμε ότι σε σχέση με το Πείραμα 1, στην pytorch το validation accuracy μεταβάλλεται πιο ομαλά και ότι έχει μεγαλύτερο training accuracy, δηλαδή και περισσότερο overfitting. Αυτό φαίνεται καθώς αυξάνεται το validation loss ενώ το training accuracy αυξάνεται και το validation accuracy μειώνεται. Λόγω άρα μεγαλύτερου overfitting, το testing accuracy είναι λίγο μικρότερο από της from scratch υλοποίησης. Σε σχέση με την αντίστοιχη from scratch υλοποίηση του πειράματος 1 παρατηρούμε ότι στην pytorch έχουμε μικρότερο χρόνο εκπαίδευσης αφού τώρα είναι 46 λεπτά. Συνολικά όμως παρατηρούμε ότι τα αποτελέσματα μεταξύ πειράματος 1 και 2 δεν απέχουν πολύ μεταξύ τους.

C. Πείραμα 3

Πραγματοποιούμε το Πείραμα 1 στην from scratch υλοποίηση αλλά με learning rate 10^{-4}:

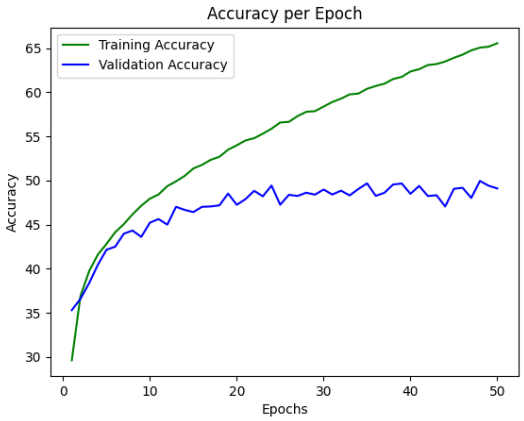


Figure 5: From Scratch, Testing Accuracy: 49.37 %

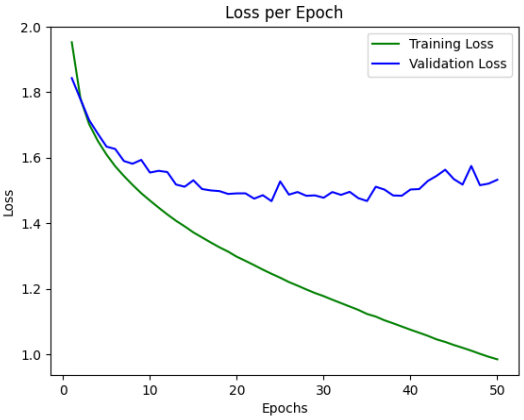


Figure 6: From Scratch

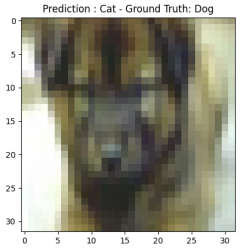
- Παρατηρούμε ότι η μείωση του learning rate προκαλεί αλλαγές στα διαγράμματα. Αρχικά για τις ίδιες εποχές αυξάνεται το testing accuracy. Τα βάρη και τα biases πρακτικά ανανεώνονται με πιο αργό ρυθμό. Ακόμη το validation accuracy αργεί να φτάσει την μέγιστη του τιμή ακριβώς λόγω του αργού αυτού ρυθμού. Όσον αφορά το loss, μειώνεται σημαντικά σε σχέση με το προηγούμενο learning rate. Τέλος ο χρόνος εκπαίδευσης του πειράματος ήταν 1 ώρα και 30 λεπτά, δηλαδή η μείωση του learning rate δεν τον επηρεάζει

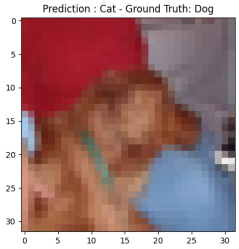
- Χαρακτηριστικά Παραδείγματα Σωστής και Λανθασμένης κατηγοριοποίησης:

  Καθώς στο πείραμα αυτό έχουμε το μεγαλύτερο testing accuracy με from scratch υλοποίηση, επιλέγουμε να παραθέσουμε για αυτό ορισμένα παραδείγματα κατηγοριοποίησης.

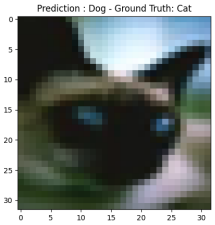
Κλάση	Testing Accuracy
0	56.8 %
1	65.4 %
2	43.6 %
3	43.2 %
4	37.7 %
5	32 %
6	49.9 %
7	53.4 %
8	67.4 %
9	44.3 %

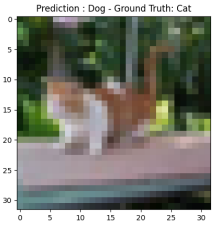
- Παρατηρούμε ότι το μικρότερο ποσοστό επιτυχίας υπάρχει στην κλάση 5 η οποία αντιστοιχεί στον σκύλο. Πραγματοποιώντας μία σχετική ανάλυση διαπιστώσαμε ότι κατά 42%, ο σκύλος συγχέεται με γάτα. Το γεγονός αυτό εξηγείται από την ομοιότητα των 2 ζώων συγκριτικά με τις υπόλοιπες κλάσεις που υπάρχουν. Παρατίθενται παρακάτω 2 εικόνες σκύλου οι οποίες κατηγοριοποιήθηκαν ως γάτες:





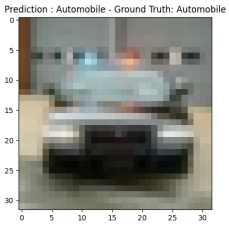
- Τώρα παρατίθενται 2 εικόνες γάτας που κατηγοριοποιήθηκαν ως σκύλοι:

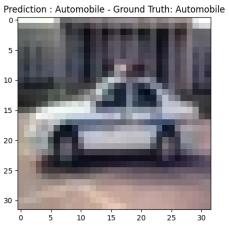


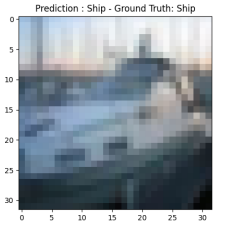


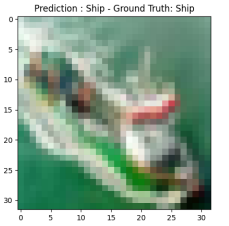
- Ακόμη από τον παραπάνω πίνακα παρατηρήθηκε πως τα μεγαλύτερα ποσοστά επιτυχημένης κατηγοριοποίησης ήταν αυτά του πλοίου και του αυτοκινήτου. Αυτό μπορεί να εξηγηθεί καθώς αποτελούν αρκετά διαχωρίσιμες κατηγορίες σε σχέση με τις

υπόλοιπες κλάσεις. Παρακάτω παρατίθενται μερικές φωτογραφίες σωστής κατηγοριοποίησης τους:









D. Πείραμα *4*

Το πείραμα αυτό είναι η εκτέλεση του πειράματος 3 αλλά στην Pytorch. Δηλαδή υπάρχουν 3 dense layers [110, 70, 70] και learning rate 10^{-4}.

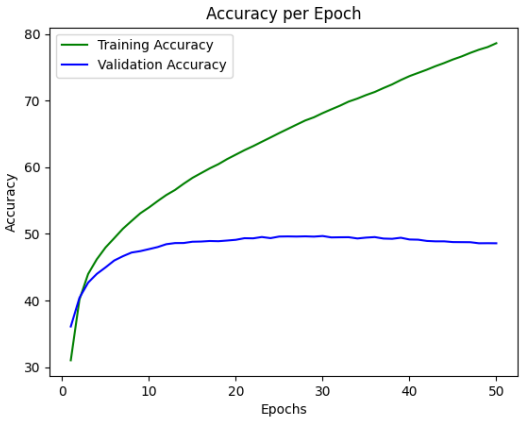


Figure 7: Pytorch, Testing Accuracy: 47.89 %

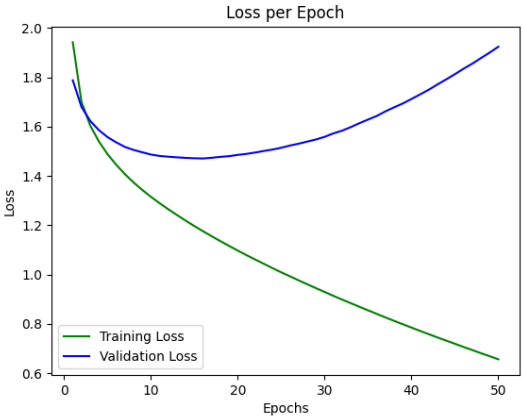


Figure 8: Pytorch

- Τα συμπεράσματα μας είναι αρκετά παρόμοια με αυτά του πειράματος 3. Αρχικά σε σχέση με το πείραμα 2 στο οποίο υπήρχε learning rate 10^{-3}, για τις ίδιες εποχές αυξήθηκε το testing accuracy. Ακόμη το accuracy αργεί να φτάσει την τελική του τιμή ακριβώς λόγω του αργού αυτού ρυθμού. Μάλιστα η μεταβολή των training και validation accuracy και loss είναι πιο ομαλή σε σχέση με το πείραμα 2. Όσον αφορά το loss, για μικρότερο learning rate, μειώνεται σημαντικά αφού η τιμή του έφτανε στο πείραμα 2 την τιμή το 3.5 και τώρα δεν ξεπερνάει το 2. Τέλος, ο χρόνος εκπαίδευσης του πειράματος ήταν 47 λεπτά.

E. Πείραμα *5*

- Στα προηγούμενα πειράματα χρησιμοποιήθηκε ο stochastic gradient descent optimizer, στον οποίο το learning rate παραμένει σταθερό. Έτσι χρησιμοποιήθηκε και ο Adam Optimizer στα ίδια layers με τα προηγούμενα πειράματα, με batch size = 1.
- Ο Adam Optimizer προσαρμόζει μεταξύ άλλων, το learning rate, το momentum και προσθέτει έναν μικρό παράγοντα στον διαιρέτη (eps) προκειμένου να αυξηθεί η σταθερότητα της διαδικασίας.

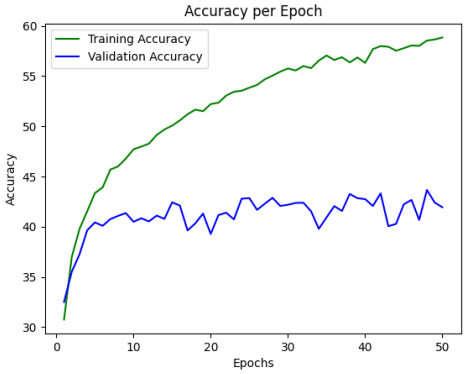


Figure 9: Pytorch, Testing Accuracy: 43.04 %

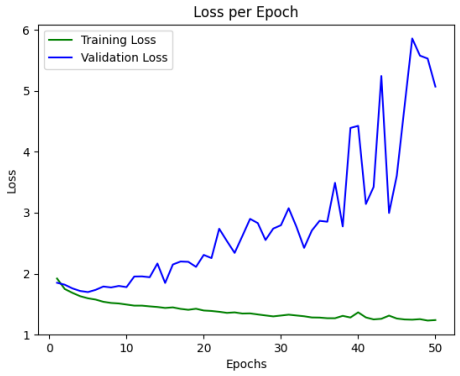


Figure 10: Pytorch

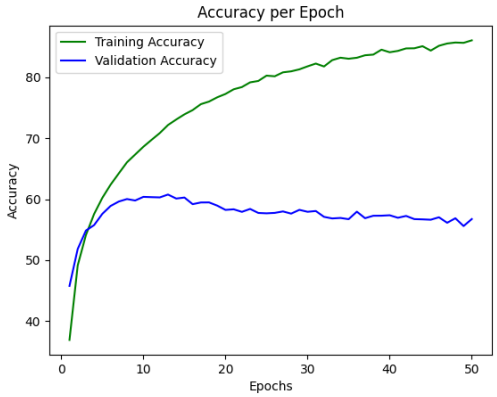


Figure 11: Pytorch, Testing Accuracy: 56.2 %

Παρατηρούμε ότι μειώνεται το training accuracy και άρα το overfiting, χωρίς όμως να αυξάνεται σημαντικά το validation accuracy. Ακόμη το loss αυξάνεται σημαντικά. Πιθανώς προκειμένου να φανούν τα πλεονεκτήματα του συγκεκριμένου optimizer να πρέπει να αλλάξουν κάποιοι παράμετροι του δικτύου καθώς και τα layers, τα οποία είναι [110, 70, 70]. Τέλος, ο χρόνος εκτέλεσης του πειράματος ήταν 2 ώρες και 9 λεπτά, δηλαδή η χρήση Adam Optimizer αυξάνει τον χρόνο εκπαίδευσης.

F. Πείραμα *6*

Για το πρώτο πείραμα με τα convolutional layers χρησιμοποιήθηκαν τα εξής layers:

- conv_1 = nn.Conv2d(3, 6, 5)
- conv_2 = nn.Conv2d(6, 16, 5)
- pool = nn.MaxPool2d(2, 2)

Ακόμη ο optimizer είναι ο SGD, learning rate = 10^{-3}, batch size = 1, και τα βάρη αρχικοποιούνται με he initialization.
Τα layers εφαρμόστηκαν ως εξής στην forward συνάρτηση:

- out = self.pool(torch.relu(self.conv_1(sample)))
- out = self.pool(torch.relu(self.conv_2(out)))
- out = out.view(-1, 16*5*5)
- Ακολουθούν τα dense layers με μεγέθη [110, 70, 70].

Όπως φαίνεται από τα παραπάνω χρησιμοποιούνται 2 convolutional layers και 2 maxpool layers. Τα MaxPool μειώνουν το μέγεθος των εικόνων καθώς απο 4 τιμές (εφόσον έχουν kernel και stride size ίσα με δύο) που προκύπτουν μετά από την εφαρμογή του convolutional layer, κρατάνε την μεγαλύτερη. Χρησιμοποιούνται καθώς μειώνουν την πολυπλοκότητα του προβλήματος, κρατώντας παράλληλα τα πιο σημαντικά στοιχεία (με την μεγαλύτερη τιμή), του δείγματος.

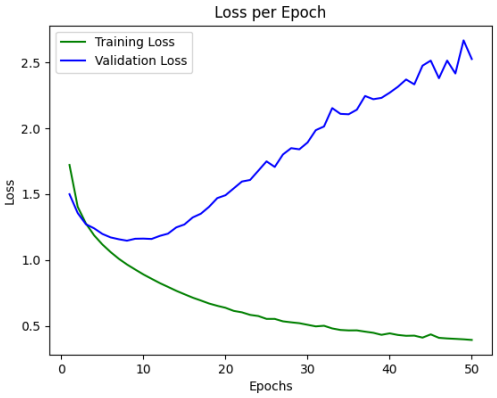


Figure 12: Pytorch

Όπως φαίνεται από τα παραπάνω διαγράμματα το validation accuracy ξεπέρασε αυτό όλων των προηγούμενων πειραμάτων. Γίνεται εμφανές ότι η χρήση convolutional layers είναι σχεδόν απαραίτητη στην κατηγοριοποίηση εικόνων. Φαίνεται βέβαια να υπάρχει μία τάση overfitting όσο περνάνε οι εποχές αφού αυξάνεται το validation loss. Στο πείραμα δηλαδή θα μπορούσε να είχε εφαρμοστεί early stopping. Τέλος ο χρόνος εκτέλεσης του πειράματος ήταν 1 ώρα και 47 λεπτά.

G. Πείραμα 7

Σε αυτό το πείραμα προσπαθούμε να εφαρμόσουμε ένα ακόμα πιο περίπλοκο Convolutional Neural Network προκειμένου να δούμε τι ποσοστά σωστής κατηγοριοποίησης μπορούμε να έχουμε. Τα layers τα οποία χρησιμοποιήσαμε είναι:

- self.conv_1 = nn.Conv2d(in_channels = 3, out_channels = 32, kernel_size = 3, padding = 1)
- self.conv_2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size = 3, padding = 1)
- self.conv_3 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size = 3, padding = 1)
- self.pool = nn.MaxPool2d(kernel_size = 2, stride = 2)

Εφαρμόστηκαν ως εξής στην forward συνάρτηση:

- out = self.relu(self.batch_norm_32(self.conv_1(sample)))
- out = self.pool(self.relu((self.batch_norm_64(self.conv_ 2(out)))))
- out = self.pool(self.relu(self.batch_norm_64(self.conv_3 (out))))
- out = out.view(out.size(0), -1)
- Ακολουθούν 2 dense layers με μεγέθη 1024, 512.

  Το input_size των dense layers αλλάζει και γίνεται: 64 x 8 x 8, το 8 λογω των MaxPool και το 64 λογω των καναλιών εξόδου του τρίτου convolutional layer. Ακόμη ο optimizer είναι ο SGD, learning rate = 10^{-3}, batch size = 1, και τα βάρη αρχικοποιούνται με he initialization.

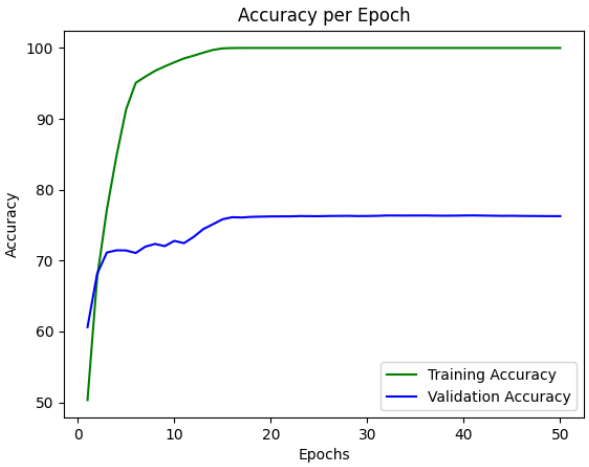


Figure 13: Pytorch, Testing Accuracy: 76.16 %

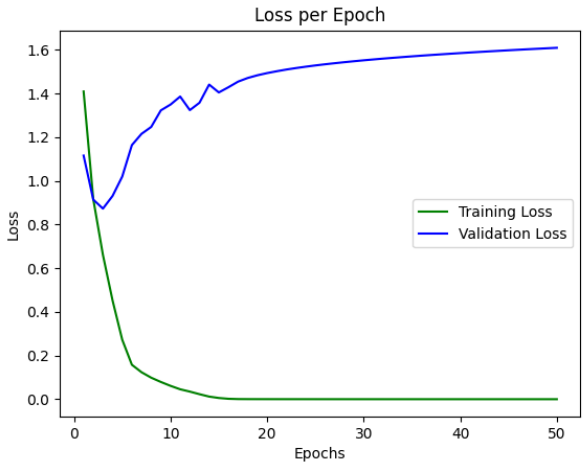


Figure 14: Pytorch

Παρατηρούμε ότι για αυτά τα layers καταφέραμε να μειώσουμε το loss αλλά και να αυξήσουμε σημαντικά το validation και testing accuracy. Το τελευταίο μάλιστα έγινε 76.16 %. Το δίκτυο φαίνεται να κάνει overfitting αφού το training accuracy έχει φτάσει το 100 % ενώ το validation accuracy παραμένει σταθερό με το validation loss να αυξάνεται. Ακόμη παρατηρούμε ότι η διαδικασία εκμάθησης θα μπορούσε να είχε τερματιστεί νωρίτερα από τις 20 εποχές, εφαρμόζοντας early stopping. Τέλος ο χρόνος εκπαίδευσης του για 50 εποχές ήταν περίπου 8 ώρες, καθιστώντας τη διαδικασία εκπαίδευσης του πολύ πιο αργή σε σχέση με αυτές των προηγούμενων πειραμάτων.

H. Πείραμα *8*

Για αυτό το πείραμα εφαρμόσαμε ένα δίκτυο που βρήκαμε στο διαδίκτυο:

- https://www.analyticsvidhya.com/blog/2021/09/convolutional-neural-network-pytorch-implementation-on

Η ανάλυση του δικτύου φαίνεται παρακάτω:

- self.conv_1 = nn.Conv2d(in_channels = 3, out_channels = 32, kernel_size = 3, padding = 1)
- self.conv_2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size = 3, padding = 1)
- self.conv_3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size = 3, padding = 1)
- self.conv_4 = nn.Conv2d(in_channels=128, out_channels = 128, kernel_size = 3, padding = 1)
- self.conv_5 = nn.Conv2d(in_channels=128, out_channels = 256, kernel_size = 3, padding = 1)
- self.conv_6 = nn.Conv2d(in_channels=256, out_channels = 256, kernel_size = 3, padding = 1)

Ακόμη:

- Το batch size είναι 1 και χρησιμοποιείται ο SGD Optimizer.
- Το learning rate είναι 10^{-3} και τα βάρη αρχικοποιούνται με he initialization.
- Χρησιμοποιείται η συνάρτηση ενεργοποίησης relu.
- MaxPool layer είναι το: nn.MaxPool2d(kernel_size = 2, stride = 2)
- Υπάρχουν 2 dropout layers, ένα 2D και ένα 1D. Αυτά είναι τα self.dropout_05 = nn.Dropout2d(p = 0.05) και self.dropout_1 = nn.Dropout(p = 0.1).

Τα layers εφαρμόζονται με την εξής σειρά στην συνάρτηση forward:

- out = self.relu(self.batch_norm_32(self.conv_1(sample)))
- out = self.pool(self.relu(self.conv_2(out)))
- out = self.relu(self.batch_norm_128(self.conv_3(out)))
- out = self.pool(self.relu(self.conv_4(out)))
- out = self.dropout_05(out)
- out = self.relu(self.batch_norm_256(self.conv_5(out)))
- out = self.pool(self.relu(self.conv_6(out)))
- out = out.view(out.size(0), -1)
- out = self.dropout_1(out)
- out = self.relu(self.input_layer(out))
- Ακολουθούν 2 dense layers με μεγέθη [1024, 512].

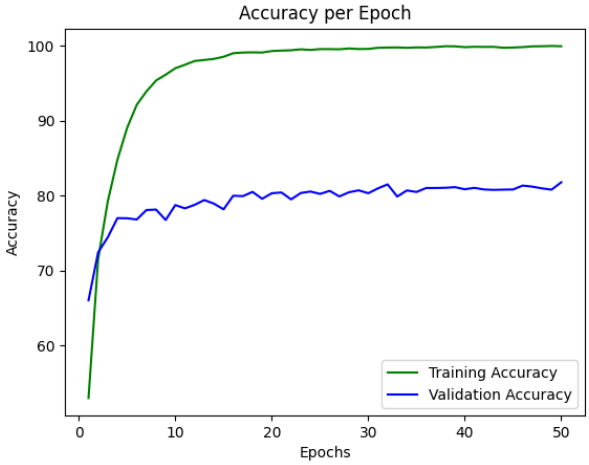


Figure 15: Pytorch, Testing Accuracy: 82.18 %

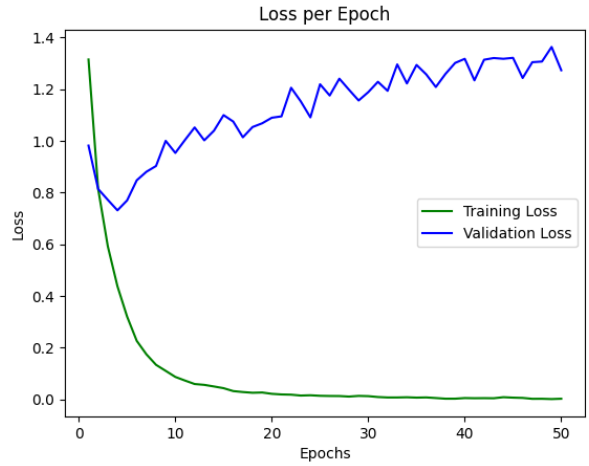


Figure 16: Pytorch

Το συγκεκριμένο νευρωνικό δίκτυο έχει τις καλύτερες επιδόσεις συγκριτικά με τα προηγούμενα. Διατηρώντας σχετικά χαμηλό validation loss καταφέρνει να αυξήσει το validation accuracy και testing accuracy. με το δεύτερο να φτάνει την τιμή 82%. Φυσικά ο χρόνος εκπαίδευσης είναι μεγαλύτερος σε σχέση με τα προηγούμενα πειράματα, καθώς για 50 εποχές πλησιάζει τις 13 ώρες.

VI. Συγκέντρωση Αποτελεσμάτων

Πείραμα	Testing Accuracy	Χρόνος Εκτέλεσης
1	47.38 %	1 ώρα : 28 λεπτά
2	45.43 %	46 λεπτά
3	49.37 %	1 ώρα : 30 λεπτά
4	47.89 %	47 λεπτά
5	43.04 %	2 ώρες : 9 λεπτά
6	56.2 %	1 ώρα : 47 λεπτά
7	76.16 %	8 ώρες : 13 λεπτά
8	82.18 %	12 ώρες : 46 λεπτά

Ο χρόνος εκτέλεσης για κάθε πείραμα αναφέρεται σε 50 εποχές. Όπως γίνεται αντιληπτό, το training με convolutional layers είναι πιο αργό αλλά και πιο αποδοτικό.

VII. KNN, Nearest Centroid

Το πρόβλημα που καλείται να αντιμετωπίσει το νευρωνικό δίκτυο που υλοποιούμε, είναι η κατηγοριοποίηση εικόνων σε κλάσεις. Εκτός από τα νευρωνικά δίκτυα, υφίστανται και άλλοι αλγόριθμοι που χρησιμοποιούνται για τον σκοπό αυτό. Δύο από αυτούς είναι ο Κατηγοριοποιητής Πλησιέστερου Γείτονα (KNN) και ο Κατηγοριοποιητής Πλησιέστερου Κέντρου (Nearest Centroid). Προκειμένου να συγκριθούν με το Νευρωνικό μας Δίκτυο, υλοποιήθηκαν σε python.

A. KNN

Ο Κατηγοριοποιητής Πλησιέστερου Γείτονα υπολογίζει για κάθε δείγμα ελέγχου, την ευκλείδεια απόσταση του (δηλαδή των συντεταγμένων του) από κάθε δείγμα εκπαίδευσης. Έπειτα επιλέγει τα k δείγματα εκπαίδευσης με την μικρότερη απόσταση από το δείγμα ελέγχου, βρίσκει τις κλάσεις των k δειγμάτων και επιλέγει αυτή που εμφανίζεται με πλειοψηφία. Σε περίπτωση ισοψηφίας επιλέγεται η κλάση του κοντινότερου δείγματος. Στο πλαίσιο δοκιμής του κώδικα με την Cifar-10, χρησιμοποιήθηκαν οι τιμές k=1 κα k=3 με τα ποσοστά επιτυχίας να είναι 35.39% και 35.61% αντίστοιχα.

B. Nearest Centroid

Ο κατηγοριοποιητής πλησιέστερου κέντρου, χωρίζει όλα τα δείγματα εκπαίδευσης στις αντίστοιχες κλάσεις τους και έπειτα υπολογίζει το κέντρο όλων των δειγμάτων που ανήκουν στην ίδια κλάση. Στη συνέχεια υπολογίζει την απόσταση του κάθε δείγματος ελέγχου από τα κέντρά των διαθέσιμων κλάσεων και το τοποθετεί στην κλάση που έχει το πλησιέστερο σε αυτό κέντρο. Η δοκιμή του αλγόριθμου nearest centroid στο σύνολο δεδομένων της Cifar-10 επέφερε 27.74% ποσοστό επιτυχίας.

C. Συμπεράσματα

Οι δύο παραπάνω αλγόριθμοι στηρίζονται στην ιδέα ότι τα πλησιέστερα δείγματα στο δείγμα ελέγχου και συγκεκριμένα η κλάση στην οποία ανήκουν, μας δίνει πληροφορίες για την κλάση του δείγματος ελέγχου. Αυτή η παραδοχή και η απλότητα του κώδικα μπορεί να εξηγήσει το γεγονός ότι έχουν χαμηλές επιδόσεις σε ένα απαιτητικό σύνολο δεδομένων. Οι αλγόριθμοι αυτοί δεν μαθαίνουν σχέσεις μεταξύ της κάθε κλάσης και των δεδομένων εισόδου, ούτε μαθαίνουν από τους λανθασμένους υπολογισμούς τους όπως τα Νευρωνικά Δίκτυα που υλοποιήθηκαν στην εργασία αυτή. Αξίζει βέβαια να σημειωθεί η διαφορά τους σε χρόνο. Ειδικά ο nearest centroid εκτελείται μέσα σε ένα λεπτό, γεγονός που τον καθιστά πολύ πιο γρήγορο από ένα περίπλοκο νευρωνικό δίκτυο. Φυσικά ακόμα και ο KNN είναι πιο γρήγορος, εφόσον παίρνει 2-3 ώρες και ένα πολύπλοκο νευρωνικό δίκτυο με convolutional layers είναι πιο αργό. Λόγω άρα της χαμηλής τους επίδοσης, δεν προτιμούνται για δύσκολα προβλήματα κατηγοριοποίησης στα οποία η ακρίβεια είναι απαραίτητη.

VIII. Παρατηρήσεις

- Για την εργασία αυτή χρησιμοποιήθηκε python 3.9.0.
- Τα αρχεία με τον κώδικα βρίσκονται στον φάκελο "Code - Assignment 1"
- Η υλοποίηση from scratch βρίσκεται στο αρχείο neural_network_from_scratch.py
- Η υλοποίηση με pytorch και dense layers είναι στο αρχείο pytorch_fc.py
- Η υλοποίηση με pytorch και convolutional layers μπορεί να βρεθεί στο αρχείο pytorch_conv.py

IX. Βιβλιογραφία

- https://www.allaboutcircuits.com/
- https://towardsdatascience.com
- https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html
- https://www.mldawn.com/back-propagation-with-cross-entropy-and-softmax/
- https://en.wikipedia.org/wiki/Nearest_centroid_classifier
- https://www.geeksforgeeks.org/ml-nearest-centroid-classifier/
- https://machinelearningmastery.com/nearest-shrunken-centroids-with-python/
- https://www.geeksforgeeks.org/k-nearest-neighbours/
- https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning
- https://www.cs.toronto.edu/~kriz/cifar.html
- https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/
- https://www.binarystudy.com/2021/09/how-to-load-preprocess-visualize-CIFAR-10-and-CIFAR-100.html