Υπολογιστική Νοημοσύνη

Επίλυση προβλήματος ταξινόμησης με χρήση Multi-layer Perceptron δικτύου

Θεόδωρος Κατζάλης
AEM: 9282
katzalis@ece.auth.gr

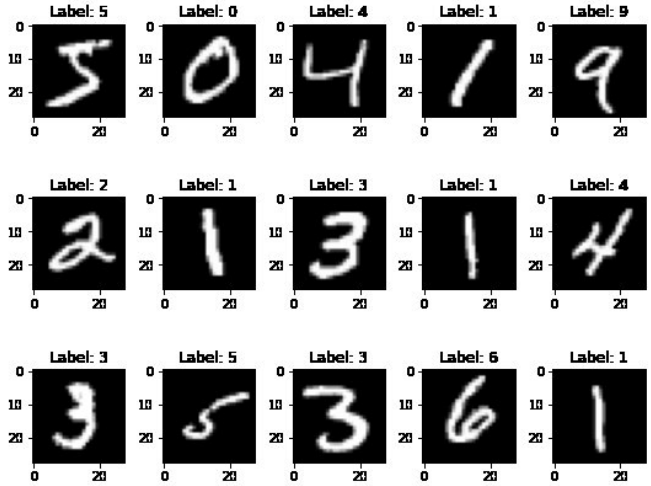
Αριστοτέλειο Πανεπιστήμιο Θεσσαλονικής

February 28, 2023

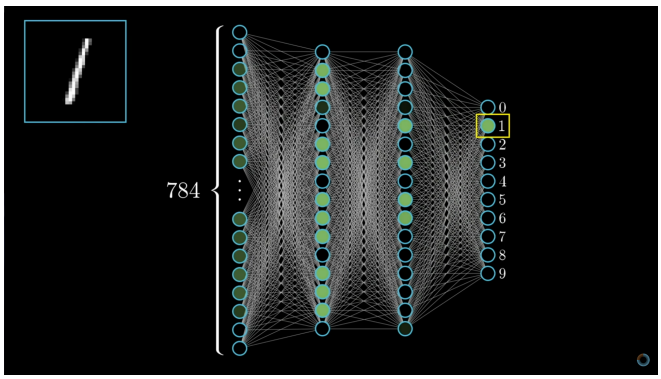
Περιεχόμενα

1 Εισαγωγή

Η συγκεκριμένη εργασία πραγματεύεται την επίλυση ενός προβλήματος ταξινόμησης (classification) χρησιμοποιώντας ένα νευρωνικό δίκτυο με αρχιτεκτονική Multi-Layer Perceptron (MLP). Στόχος είναι η κατανόηση της επίδρασης των παραμέτρων του δικτύου με απώτερο σκοπό την βελτιστοποίηση της απόδοσης του. Το dataset στο οποίο έγινε η μελέτη είναι το λεγόμενο MNIST dataset, το οποίο αποτελείται απο greyscale εικόνες 28x28 pixels που απεικονίζουν χειρόγραφα ψηφία από το 0 έως το 9. Η υλοποίηση της εργασίας έγινε με την βιβλιοθήκη `Keras`.



(a) Σύνολο δειγμάτων του MNIST (source)



(b) Απεικόνιση ενός MLP δικτύου (source)

Σχήμα 1

Σε όλη την έκταση της εργασίας, αν δεν έχει ειπωθεί διαφορετικά, η βασική δομή της αρχιτεκτονικής MLP αποτελείται απο τα εξής χαρακτηριστικά:

- Κανονικοποίηση (normalization) εισόδου min-max
- Δύο κρυφά στρώματα με προεπιλεγμένο αριθμό νευρώνων 128 και 256 αντίστοιχα
- Είσοδος δικτύου με 28*28=784 νευρώνες, μετατρέποντας την εικόνα σε διάνυσμα (row-wise)
- Συνάρτηση ενεργοποίησης κρυφών στρωμάτων `ReLU`
- Έξοδος με 10 νευρώνες, υπεύθυνοι να υποδεικνύουν την πιθανότητα πρόβλεψης κάθε ψηφίου
- Συνάρτηση ενεργοποίησης της εξόδου `softmax`
- Συνάρτηση κόστους `categorical-crossentropy`
- 20% παρακράτηση δεδομένων εκπαίδευσης για λόγους validation ($N_{train} = 4800, N_{validation} = 1200, N_{test} = 1000$)
- Προεπιλεγμένος αριθμός `batch size` 256
- Μετρική αξιολόγησης Accuracy $= \frac{\text{True Positive+True Negative}}{\text{True Positive+True Negative+False Positive+False Negative}}$

  Η πρόβλεψη του ψηφίου καθορίζεται από τον νευρώνα που έχει την μεγαλύτερη τιμή-πιθανότητα (`argmax`).

Στη συνέχεια, η ανάλυση αποτελείται απο δύο τμήματα. Στο πρώτο θα εστιάσουμε την προσοχή μας σε παραμέτρους που αφορούν την επιλογή βελτιστοποίησης (optimizer), κανονικοποίησης (regularization) και αρχικοποίησης των συναπτικών βαρών (initialization). Στο δεύτερο, θα επαναλάβουμε την εκπαίδευση του δικτύου εξερευνώντας διαφορετικές τιμές για συγκεκριμένες υπερπαραμέτρους (πχ. αριθμό νευρώνων των κρυφών στρωμάτων) χρησιμοποιώντας `grid search` με στόχο την βέλτιστη ρύθμιση του δικτύου (fine tuning).

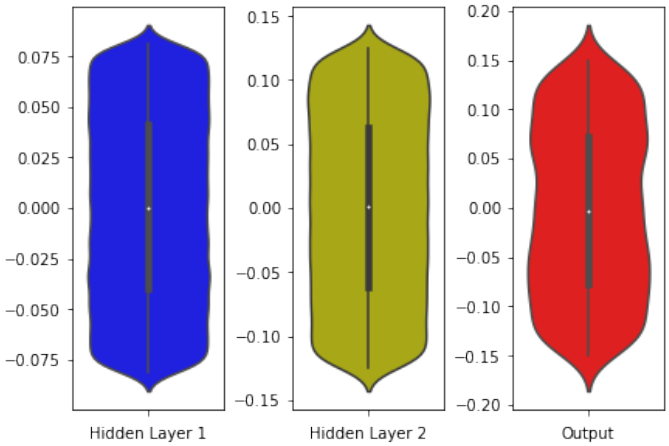
2 Διερεύνηση απόδοσης μοντέλου με διαφοροποίηση στο σχεδιασμό και τη διαδικασία εκπαίδευσης

Ως προς το πρώτο κομμάτι της εργασίας, θα δοκιμάσουμε τους συνδυασμούς που ακολουθούν ως υποενότητες. Για να διευκολύνουμε την δυνατότητα αλλαγής ορισμένων παραμέτρων κρατώντας σταθερή την γενικότερη δομή του δικτύου που αναφέρθηκε προηγουμένως, έχουμε δημιουργήσει τις συναρτήσεις (`fitWrapper()`, `create_model()`). Μέσω αυτών αλλάζουμε μόνο τα keyword arguments με τις επιθυμητές παραμέτρους που θέλουμε να πειραματιστούμε.

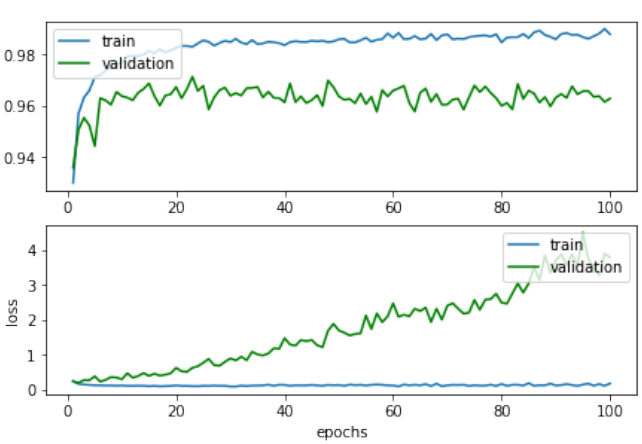
2.1 1ος συνδυασμός

Εκπαίδευση δικτύου με default παραμέτρους μεταβάλλοντας τον αριθμό του `batch_size` με διαθέσιμες τιμές [1, 256, N_{train} = 4800]. Στον συγκεκριμένο συνδυασμό χρησιμοποιούμε ως default optimizer τον `adam`.

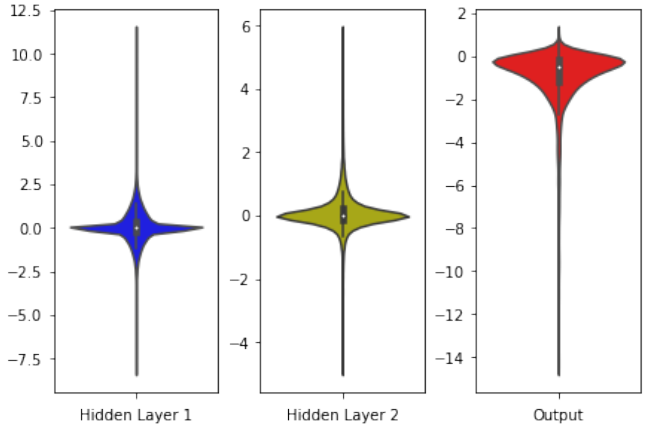
```
model = create_model()
batches = [1, 256, num_samples_training]
for batch in batches:
    model.fit(train_x, train_y, batch_size=batch, epochs=100, validation_split=0.2, shuffle=False)
```



Σχήμα 2: Violin διάγραμμα των συναπτικών βαρών πριν την εκπαίδευση

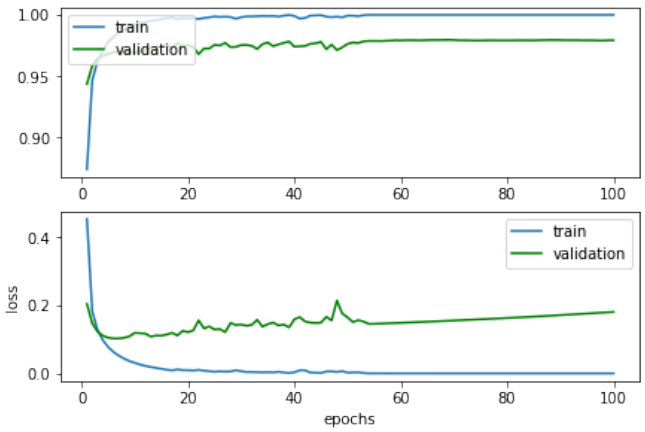


(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος

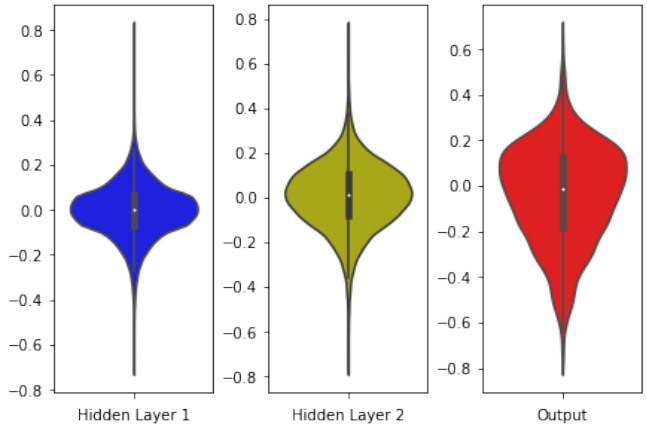


(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 3: Batch size = 1 (online)

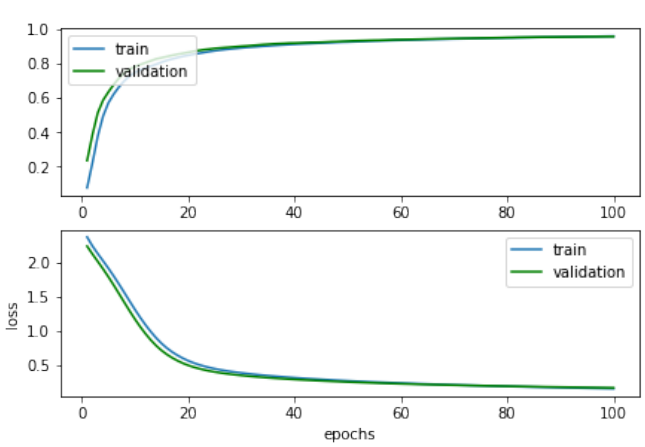


(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος

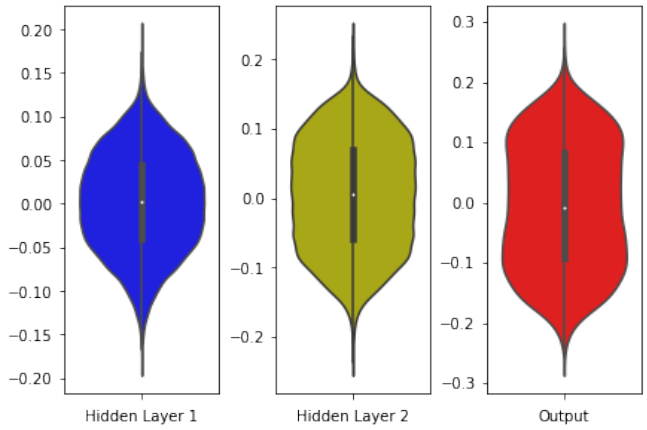


(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 4: Batch size = 256 (minibatch)



(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος



(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 5: Batch size = N_{train}

batch_size / metrics	1	256	$N_{train} = 4800$
Accuracy (%)	96.10	98.17	95.48
Time Elapsed (s)	8913	108	55
Weight update iterations	$\frac{4800}{1} = 4800 * 100$	$\frac{4800}{256} = 188 * 100$	$\frac{4800}{4800} = 1 * 100$

Πίνακας 1: Χρόνος εκπαίδευσης και η ακρίβεια του μοντέλου στο testing υποσύνολο

Συμπερασματικά, η εκπαίδευση γίνεται ολοένα και πιο αργή όσο μειώνεται το `batch size`. Η εφαρμογή των minibatch συνήθως αποτελεί καλή πρακτική για την εκπαίδευση των νευρωνικών δικτύων διότι έχουν καλύτερη ικανότητα `generalization`, απαιτείται λιγότερη μνήμη αλλά και σε ειδικές περιπτώσεις όπως με χρήση optimizer stochastic gradient descent, υπάρχει η δυνατότητα αποφυγής "κακών" τοπικών ελαχίστων. Ο χρόνος εκπαίδευσης σε μικρότερα `batches` οφείλεται στην αύξηση των αριθμών της ανανέωσης των συναπτικών βαρών για δεδομένο αριθμό εποχών (epochs). Έτσι για κάθε εποχή, θα έχουμε $\frac{N_{train}}{batch\_size}$ επαναλήψεις.

Στην περίπτωση του online batch παρατηρείται μια σταθερή ανοδική πορεία του validation loss μετά απο περίπου 10 εποχές, ενώ το validation accuracy παραμένει περίπου σταθερό. Αυτό σημαίνει ότι με την αύξηση των εποχών, το δίκτυο γίνεται ολοένα και λιγότερο σίγουρο για την πρόβλεψη των ψηφίων με αποτέλεσμα να αυξάνεται το loss, αλλά το accuracy να παραμένει στα ίδια επίπεδα μιας και η τελική επιλογή του δικτύου είναι ο νευρώνας με την μέγιστη πιθανότητα. Βέβαια απο την στιγμή που το validation loss αρχίζει και αποκλίνει απο το training loss, αυτό αποτελεί σημάδι overfitting.

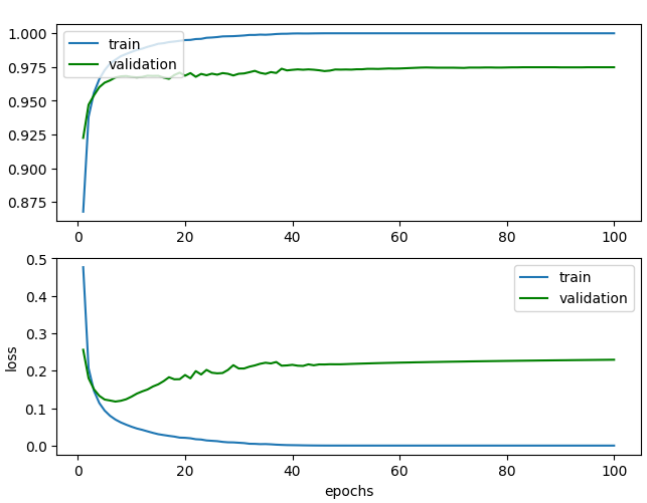
Overfitting παρατηρείται ακόμη και στην περίπτωση του minibatch, στις πρώτες κιόλας εποχές, εξαιτίας της χαρακτηριστικής απόκλισης του validation loss απο το training loss. Αξίζει βέβαια να σημειωθεί ότι στις περιπτώσεις online και minibatch, γίνονται 4800 και 188 ανανεώσεις αντίστοιχα των συναπτικών βαρών σε κάθε εποχή. Το διάγραμμα θα ήταν διαφορετικό αν στον άξονα x είχαμε τις επαναλήψεις ανανέωσης των βαρών.

Για την τελευταία περίπτωση του batch_size (N_{train}) έχοντας μια ανανέωση βαρών ανα εποχή, δεν παρατηρείται κάποιο φαινόμενο overfitting. Το loss και το accuracy του training παρουσιάζει όμοια εξέλιξη με τα αντίστοιχα μεγέθη του validation, το οποίο αφήνει περιθώρια περαιτέρω εκπαίδευσης (underfitting) μιας και το training error δεν είναι τόσο χαμηλό όσο στις προηγούμενες περιπτώσεις αλλά και το accuracy παρουσιάζει μια μικρή μείωση.

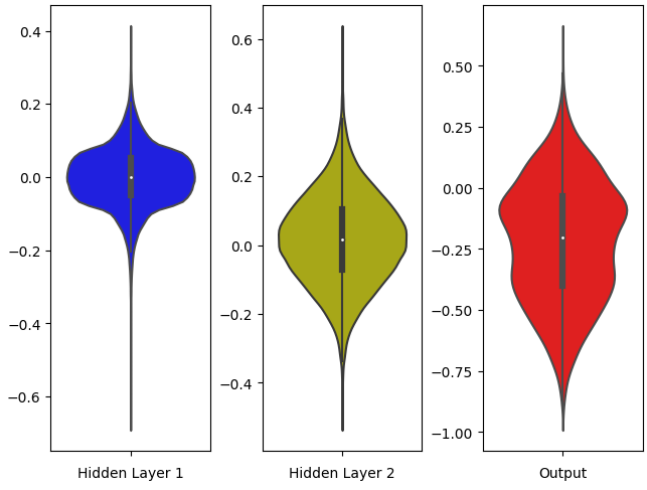
2.2 2ος συνδυασμός

Εκπαίδευση δικτύου με optimizer `RMSProp` (`learning rate` = 0.001, `rho` $\in \{0.01, 0.99\}$). Επιλέγεται απο εδώ και στο εξής ως `batch size` η προεπιλεγμένη τιμή 256.

```
rhos = [0.01, 0.9]
for rho in rhos:
    model = create_model(optimizer=RMSprop(learning_rate=0.001, rho=rho))
    model.fit(train_x, train_y, batch_size=256, epochs=100, validation_split=0.2, shuffle=False)
```

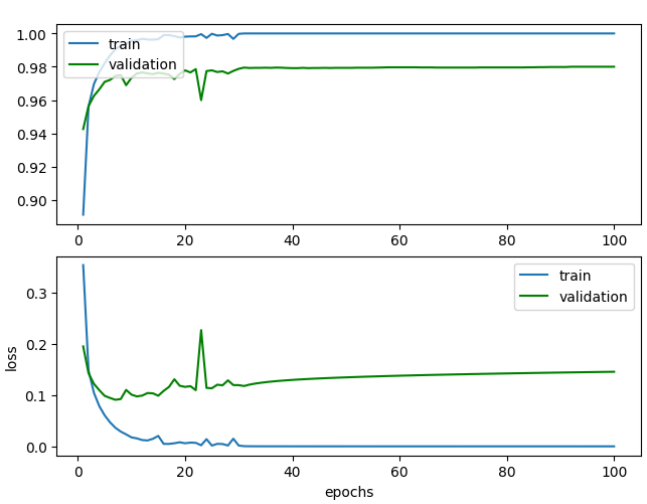


(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος

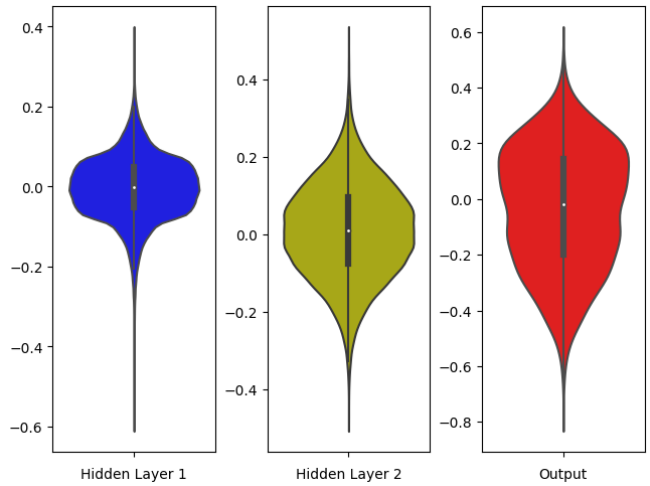


(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 6: rho = 0.01



(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος



(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 7: rho = 0.99

metrics \ rho	0.01	0.99
Accuracy (%)	97.72	98.2
Loss	0.2	0.11

Πίνακας 2: Η ακρίβεια του μοντέλου και το τελικό loss στο testing υποσύνολο

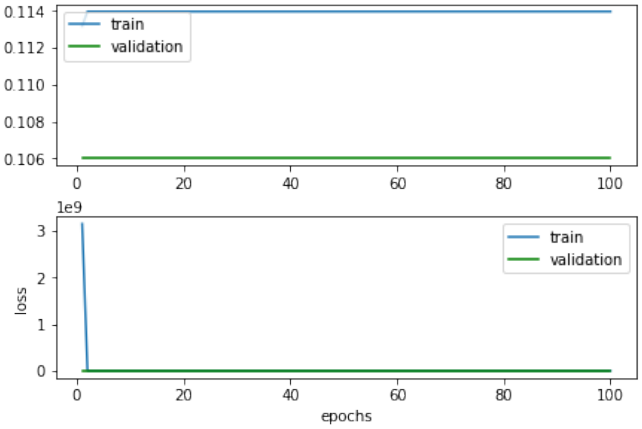
Συμπερασματικά, σχετικά με την διαφορά στην επιλογή της τιμής του rho, δεν παρατηρείται κάποια σημαντική αλλαγή στην τελική απόδοση του μοντέλου. Ωστόσο παρατηρείται η περίπτωση του overfitting απο τις πρώτες κιόλας εποχές.

Από εδώ και στο εξής για τους παρακάτω συνδυασμούς που χρησιμοποιούμε τον συγκεκριμένο optimizer, θα επιλέγουμε `rho=0.99`.

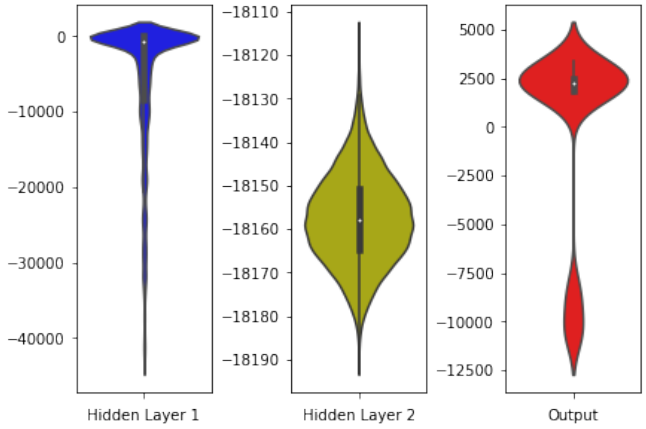
2.3 3ος συνδυασμός

Εκπαίδευση δικτύου με αρχικοποίηση βαρών (εκ προεπιλογής το keras χρησιμοποιεί glorot unifrom initialization) επιλέγοντας κανονική κατανομή με μέσο όρο 10 και optimizer stochastic gradient descent.

```
model = create_model(optimizer="sgd", kernel_initializer=RandomNormal(mean=10))
model.fit(train_x, train_y, batch_size=256, epochs=100, validation_split=0.2, shuffle=False)
```



(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος



(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 8: loss = 2.30, accuracy = 11.34%

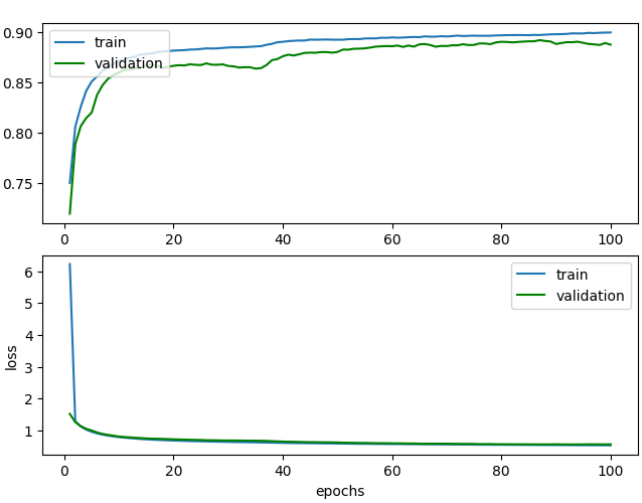
Σε αυτόν τον συνδυασμό είναι χαρακτηριστική η επίπτωση της αρχικοποίησης των συναπτικών βαρών. Ξεκινήσαμε από ένα πολύ "κακό" σημείο (πολύ μακρυά από την τελική βέλτιστη τιμή) εξαιτίας αυτής της αρχικοποίησης, μέσω του οποίου προσπαθούμε να βρούμε το μονοπάτι ελαχιστοποίησης του σφάλματος. Αποτέλεσμα αυτού είναι μια αρκετά χαμηλή ακρίβεια για το μοντέλο μας.

2.4 4ος συνδυασμός

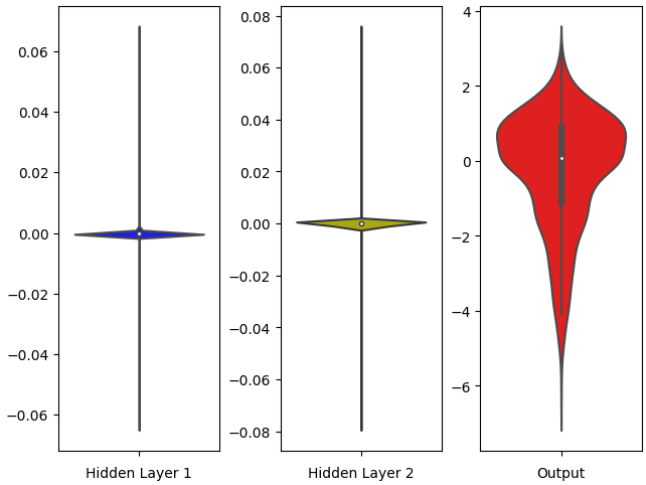
2.4.1 RMSrop

Εκπαίδευση δικτύου με optimizer `RMSprop` και προσθήκη regularization L_2 με $\alpha \in \{0.1, 0.01, 0.001\}$ στα κρυφά στρώματα.

```
alphas = [0.1, 0.01, 0.001]
for alpha in alpha:
    model = create_model(optimizer=RMSprop(learning_rate=0.001, rho=0.9), kernel_regularizer=l2(alpha)
    model.fit(train_x, train_y, batch_size=256, epochs=100, validation_split=0.2, shuffle=False)
```

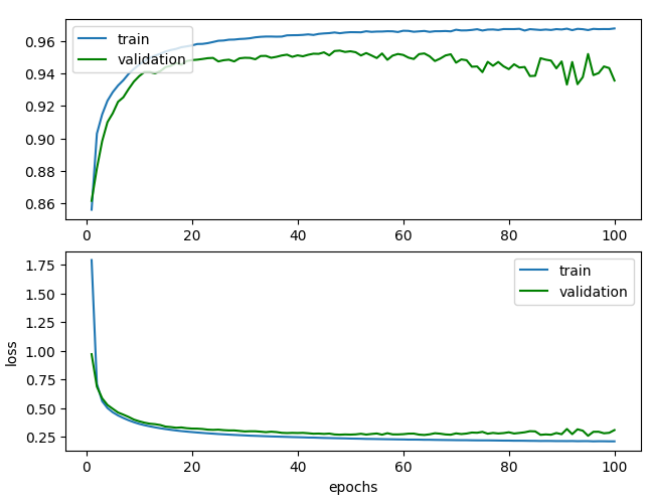


(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος

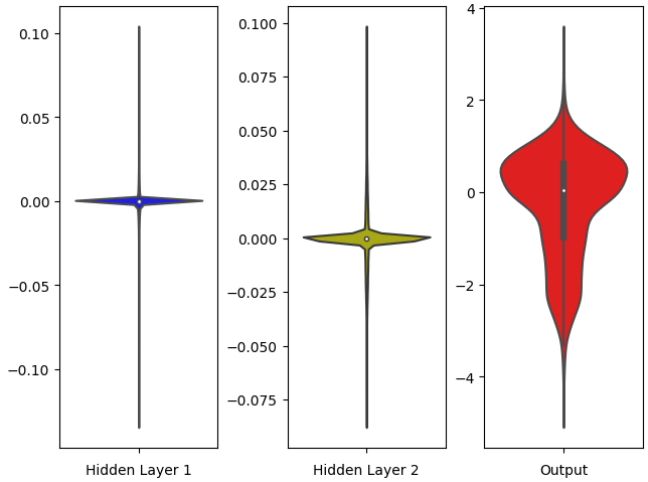


(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 9: $\alpha = 0.1$

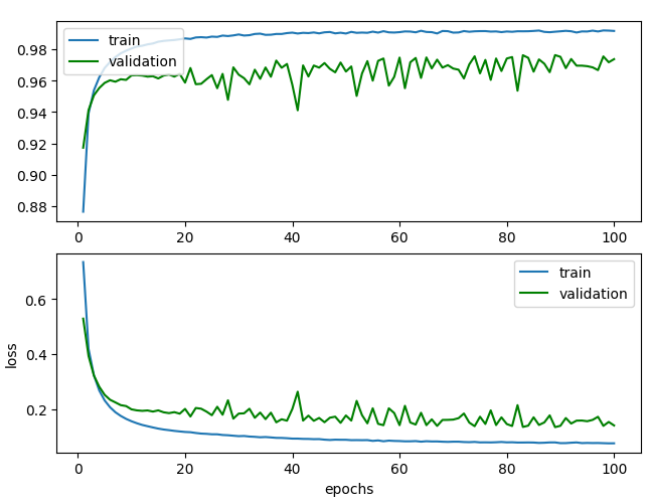


(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος

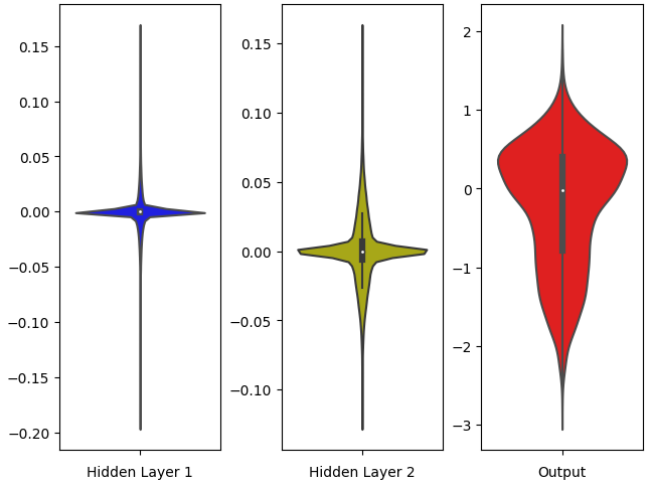


(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 10: $\alpha = 0.01$



(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος



(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 11: $\alpha = 0.001$

learning_rate / metrics	0.1	0.01	0.001
Accuracy (%)	88.37	93.45	97.40
Loss	0.56	0.3	0.13

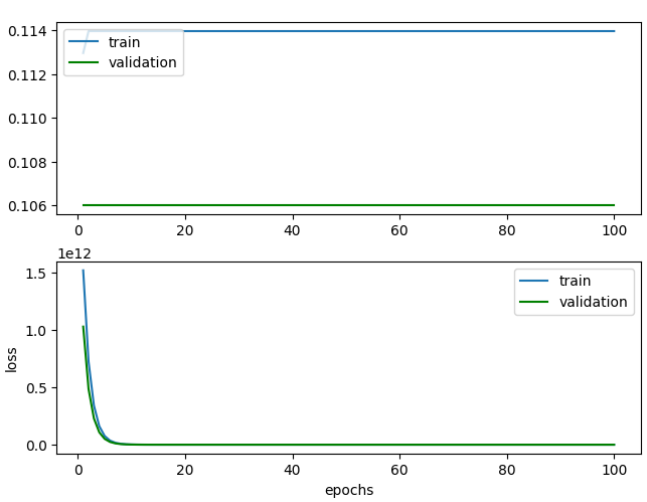
Πίνακας 3: Η ακρίβεια του μοντέλου και το τελικό loss στο testing υποσύνολο

Παρατηρούμε ότι με την προσθήκη κανονικοποίησης, εξαλείφονται φαινόμενα overfitting και οι καμπύλες εκμάθησης ακολουθούν την ίδια πορεία χωρίς να αποκλίνει σημαντικά η μια από την άλλη. Έτσι θα μπορούσαμε να ισχυριστούμε ότι το μοντέλο μας έχει καλύτερη απόδοση γενικοποίησης, ωστόσο η τελική ακρίβεια δεν βελτιώθηκε, μιας και ήταν εξαρχής ιδιαίτερα υψηλή, εξαιτίας πιθανόν της απλότητας του dataset.

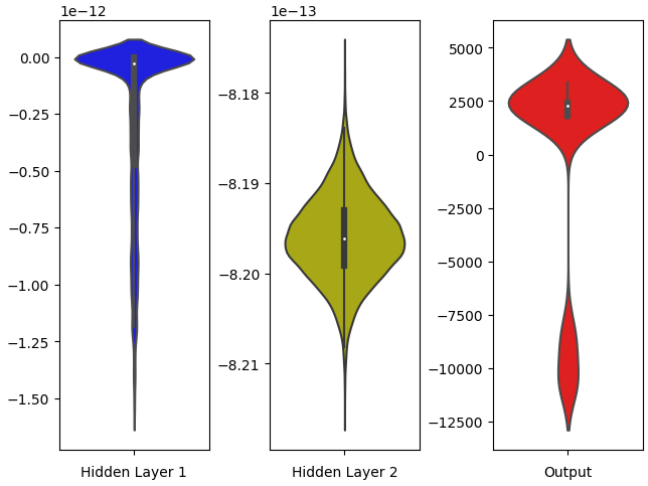
2.4.2 SGD

Εκπαίδευση δικτύου με optimizer `Stochastic Gradient Descent`, αρχικοποίηση βαρών με κανονική κατανομή (μέσο όρο 10) και προσθήκη regularization L_2 με $\alpha \in \{0.1, 0.01, 0.001\}$ στα κρυφά στρώματα.

```
alphas = [0.1, 0.01, 0.001]
for alpha in alphas:
    model = create_model(
        optimizer=SGD(lr=0.01),
        kernel_initializer=RandomNormal(10),
        kernel_regularizer=l2(alpha))
    model.fit(train_x, train_y, batch_size=256, epochs=100, validation_split=0.2, shuffle=False)
```

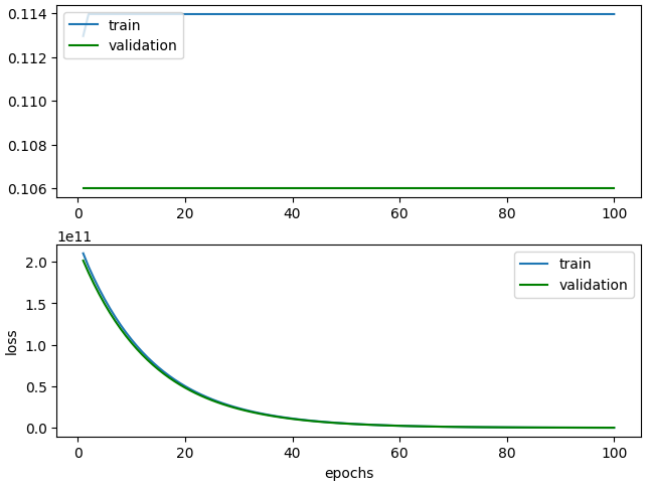


(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος

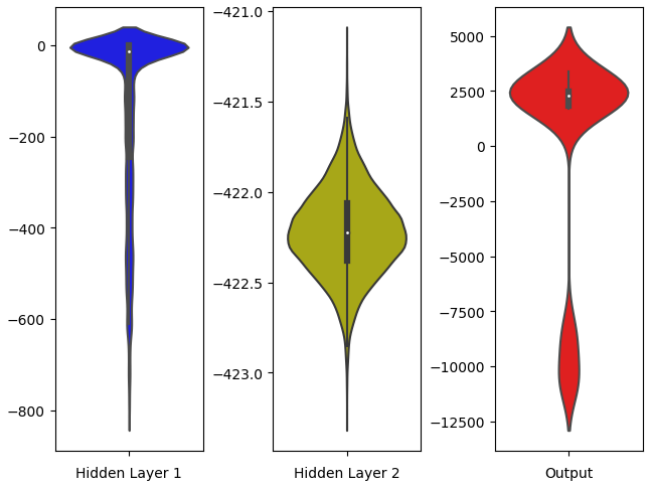


(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 12: $\alpha = 0.1$

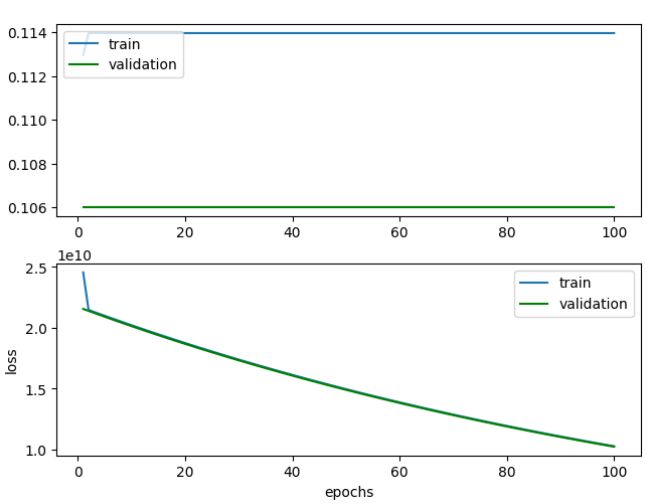


(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος

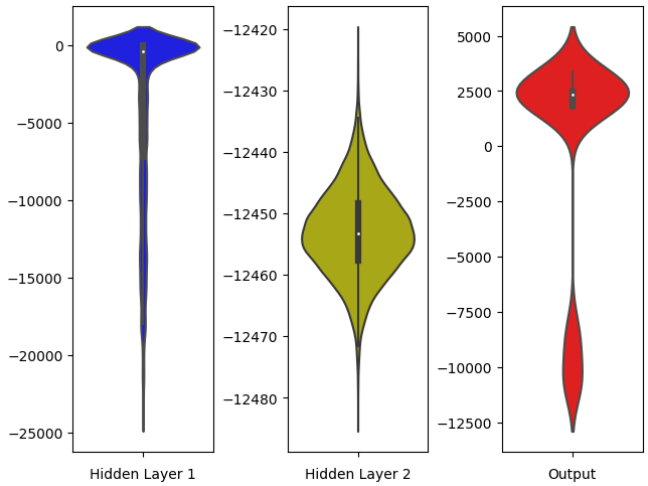


(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 13: $\alpha = 0.01$



(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος



(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 14: $\alpha = 0.001$

learning_rate / metrics	0.1	0.01	0.001
Accuracy (%)	11.34	11.34	11.35
Loss	2.3	1.17 * 1e9	102.23 * 1e9

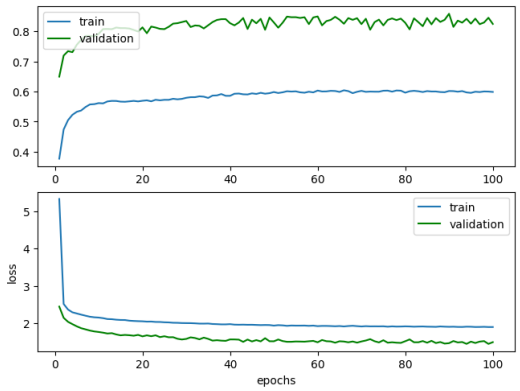
Πίνακας 4: Η ακρίβεια του μοντέλου και το τελικό loss στο testing υποσύνολο

Η ακρίβεια του μοντέλου μας είναι πολύ χαμηλή εξαιτίας της αναποτελεσματικής αρχικοποίησης των συναπτικών βαρών όπως και στο 2.3.

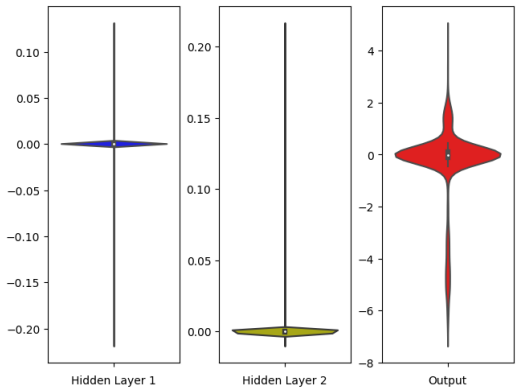
2.5 5ος συνδυασμός

2.5.1 RMSrop

```
model = create_model(
    optimizer=RMSprop(learning_rate=0.001, rho=0.99),
    kernel_regularizer=l1(0.01),
    is_dropout=True,
)
model.fit(train_x, train_y, batch_size=256, epochs=100, validation_split=0.2, shuffle=False
```



(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος



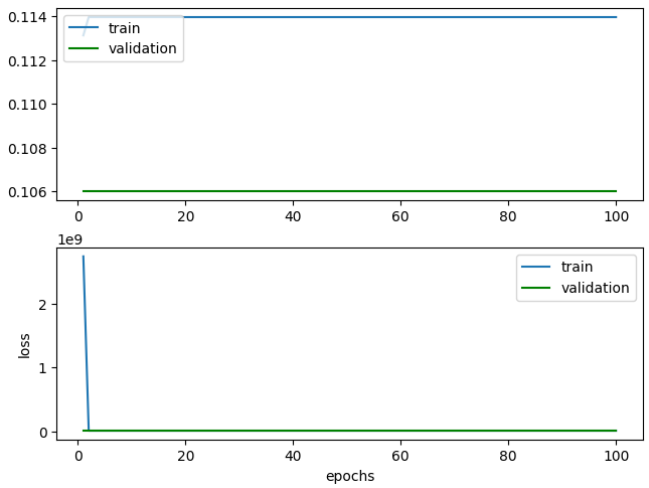
(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 15: loss=1.49, accuracy=82.26%

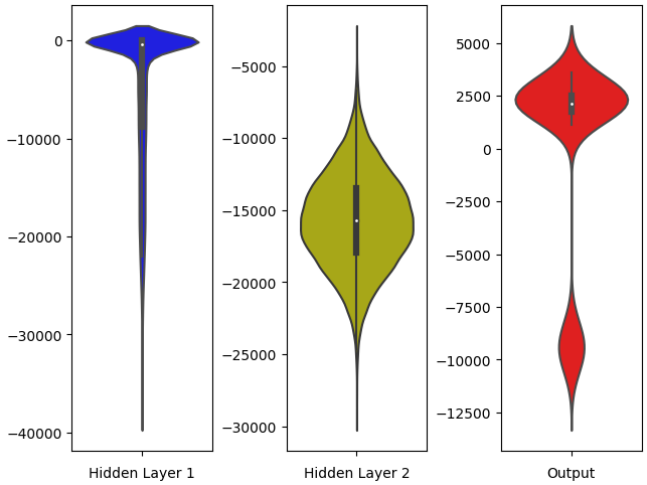
Εδώ παρατηρούμε φαινόμενο underfitting. Η κανονικοποίηση κρατάει σε αρκετά χαμηλά επίπεδα τις τιμές των βαρών, με αποτέλεσμα το δίκτυο μας να μην μαθαίνει γρήγορα. Το φαινόμενο underfitting εξηγείται από τις χαμηλές τιμές του training acccuracy αλλά και από την σταθερή πορεία της απώλειας του training set.

2.5.2 SGD

```
model = create_model(
    optimizer=SGD(lr=0.01),
    kernel_initializer=RandomNormal(10),
    kernel_regularizer=l1(0.01),
    is_dropout=True,
)
model.fit(train_x, train_y, batch_size=256, epochs=100, validation_split=0.2, shuffle=False
```



(a) Καμπύλες εκμάθησης ακρίβειας και σφάλματος



(b) Violin διάγραμμα των συναπτικών βαρών μετά την εκπαίδευση

Σχήμα 16: loss=102 * 1e6, accuracy=11.34%

Γνωρίζοντας το πρόβλημα της αρχικοποίησης των συναπτικών βαρών (2.3), παρατηρούμε ότι με την χρήση της κανονικοποίησης δεν υπάρχει κάποια βελτίωση στην απόδοση του μοντέλου. Έτσι αποδεικνύεται πόσο σημαντική είναι η αφετηρία των τιμών των συναπτικών βαρών.

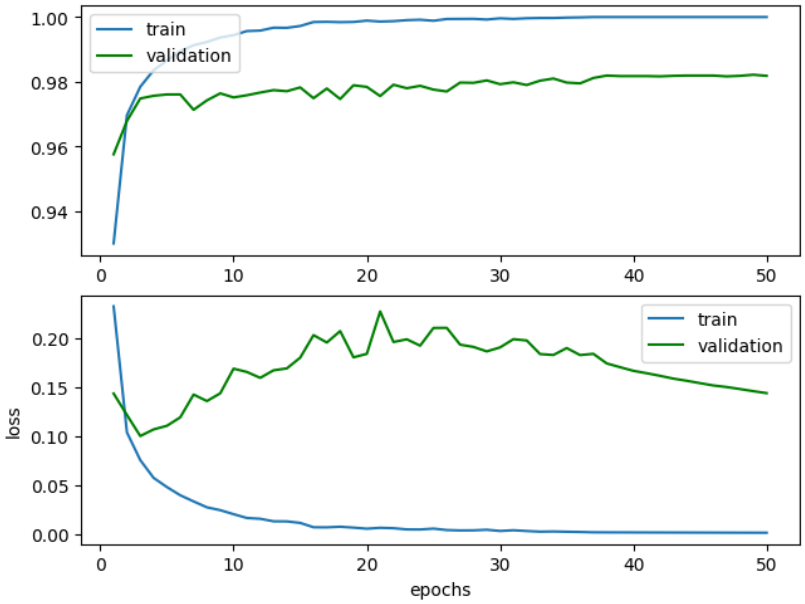
3 Fine tuning

Όπως αναφέρθηκε στην αρχή, έγιναν κάποιες συμβάσεις για κάποιες παραμέτρους του δικτύου μας όπως ο αριθμός των νευρώνων στα κρυφά στρώματα. Αυτές οι τιμές, λοιπόν, θα αποτελέσουν σημείο μελέτης σε αυτό το κομμάτι της εργασίας με σκοπό να επιτύχουμε τις βέλτιστες ως προς την ακρίβεια του μοντέλου μας. Στο δίκτυο μας θα χρησιμοποιήσουμε μεταξύ των άλλων, 2 κρυφά στρώματα, αλγόριθμο βελτιστοποίησης RMSprop και κανονικοποίηση L2[1]. Έτσι για την ρύθμισή του, θα μας απασχολήσουν α) οι τιμές για τους νευρώνες στο πρώτο και το δεύτερο κρυφό στρώμα ($n_{h1} \in \{64, 128\}, n_{h2} \in \{256, 512\}$), β) η παράμετρος κανονικοποίησης $\alpha \in \{10^{-1}, 10^{-3}, 10^{-6}\}$ και γ) ο ρυθμός εκμάθησης $lr \in \{10^{-1}, 10^{-2}, 10^{-3}\}$. Για την εξερεύνηση αυτών των παραμέτρων κάναμε χρήση του `HyperBand()` του πακέτου `keras_tuner`.

Αξίζει να σημειωθεί ότι τρέξαμε την ανάλυση για δύο διαφορετικά κριτήρια αξιολόγησης, το `val_accuracy` και `val_f1_score`, όπου στο δεύτερο χρησιμοποιήθηκε custom υλοποίηση του F1-score, με στόχο την μεγιστοποίηση του. Και στις δύο περιπτώσεις βρήκαμε τις ίδιες βέλτιστες παραμέτρους, οι οποίες είναι:

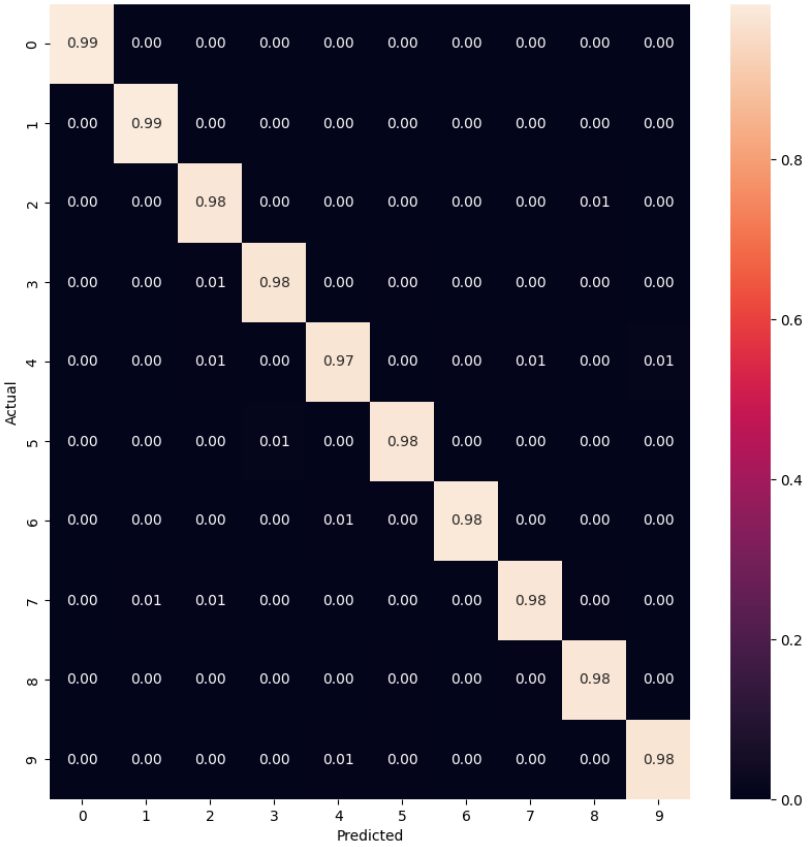
$$n_{h1} = 128$$
$$n_{h2} = 256$$
$$\alpha = 10^{-6}$$
$$lr = 10^{-3}$$

Στη συνέχεια με αυτές τις βέλτιστες υπερπαραμέτρους, προπονήσαμε το τελικό μας μοντέλο για 50 εποχές και έχουμε τα ακόλουθα αποτελέσματα:



Σχήμα 17: Καμπύλες εκμάθησης ακρίβειας και απώλειας

[1]Η ακριβής περιγραφή του μοντέλου φαίνεται από τον κώδικα που έχουμε παραθέσει στο τέλος της αναφοράς.



Σχήμα 18: Πίνακας σύγχυσης καλύτερου μοντέλου

$$accuracy = 98.29\%$$

$$F_1 = 0.9831$$

Σχετικά με τις καμπύλες εκμάθησης, παρατηρούμε το φαινόμενο overfitting απο τις πρώτες κιόλας εποχές. Ωστόσο το μοντέλο μας, με τις υπερπαραμέτρους που θέσαμε και με σύντομη προπόνηση έχει πολύ καλή απόδοση και φτάνει πολύ υψηλές τιμές ακρίβειας όπως φαίνεται και απο τον πίνακα σύγχυσης.

4 Python

In [ ]:
```
# set the seed to get reproducible results
from black import out
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from tensorflow import keras
from tensorflow.keras.optimizers import RMSprop
from keras_tuner import Objective
from keras.layers import Flatten, Dense, Dropout
from keras import Sequential
from keras.initializers import RandomNormal, HeNormal
from keras.datasets import mnist
from keras.regularizers import l2, l1
from keras.optimizers import SGD
import keras_tuner as kt
import matplotlib.pyplot as plt
import random as python_random
import tensorflow as tf
import numpy as np
import os
from pytictoc import TicToc  # time difference

# visualization
import matplotlib.pyplot as plt
import seaborn as sns


def set_seed(seed):
    os.environ["PYTHONHASHSEED"] = str(seed)
    os.environ["TF_CUDNN_DETERMINISTIC"] = str(seed)

    # source: https://keras.io/getting_started/faq/#how-can-i-obtain-reproducible-res
    # source: https://github.com/keras-team/keras/issues/2743
    np.random.seed(seed)
    python_random.seed(seed)
    tf.random.set_seed(seed)
```

In [ ]:
```
# Data preparation

#  load and normalize dataset
(train_x, train_y), (test_x, test_y) = mnist.load_data()

# vectorize image row-wise
# shape = (num_samples, num_features)
train_x = train_x.reshape(train_x.shape[0], -1)
test_x = test_x.reshape(test_x.shape[0], -1)

num_samples_training = train_x.shape[0]
num_features = train_x.shape[1]
num_classes = 10

# memory efficient
train_x = train_x.astype("float32")
test_x = test_x.astype("float32")

# min-max normalization
train_x = train_x / 255
test_x = test_x / 255

# manual one-hot encoding instead of using 'sparse_categorical_entropy' (now 'categor
# reason: https://stackoverflow.com/questions/49019383/keras-precision-and-recall-is-
train_y_1hot = keras.utils.to_categorical(train_y, num_classes)
test_y_1hot = keras.utils.to_categorical(test_y, num_classes)
```

In [ ]:
```
# Helper functions
```

```python
# build the MLP
# Notes:
# regularization can be used in the output layer too, although in most examples they
# dropout should not be used for input and output layers
def create_model(
    hidden_layer_nodes1=128,
    hidden_layer_nodes2=256,
    kernel_initializer=None,
    kernel_regularizer=None,
    is_dropout=False,
    optimizer="adam",
    metrics=["accuracy"],
):
    hidden_layer_options = {}
    output_layer_options = {}
    if kernel_initializer:
        hidden_layer_options["kernel_initializer"] = kernel_initializer
        output_layer_options["kernel_initializer"] = kernel_initializer
    if kernel_regularizer:
        hidden_layer_options["kernel_regularizer"] = kernel_regularizer

    model = Sequential()
    # 1st hidden layer
    model.add(Dense(hidden_layer_nodes1, input_shape=(num_features,), activation="rel

    if is_dropout:
        # source: https://machinelearningmastery.com/how-to-reduce-overfitting-with-d
        model.add(Dropout(0.3))

    # 2nd hidden layer
    model.add(Dense(hidden_layer_nodes2, activation="relu", **hidden_layer_options))

    if is_dropout:
        model.add(Dropout(0.3))

    # output
    model.add(Dense(num_classes, activation="softmax", **output_layer_options))
    # model.add(Dense(num_classes, activation="softmax", **hidden_layer_options))

    model.compile(optimizer=optimizer, loss="categorical_crossentropy", metrics=metri
    return model


def plot_weights(weight):
    plt.figure(constrained_layout=True)
    plt.subplot(131)
    sns.violinplot(y=weight[0], color="b")
    plt.xlabel("Hidden Layer 1")
    plt.subplot(132)
    sns.violinplot(y=weight[1], color="y")
    plt.xlabel("Hidden Layer 2")
    plt.subplot(133)
    sns.violinplot(y=weight[2], color="r")
    plt.xlabel("Output")


def filter_weights(model):
    weights = [
        model.get_weights()[0].flatten().reshape(-1, 1),  # hidden layer 1
        model.get_weights()[2].flatten().reshape(-1, 1),  # hidden layer 2
        model.get_weights()[4].flatten().reshape(-1, 1),  # output
    ]
    return weights
```

```
def fitWrapper(batch_size, epochs):
    history = model.fit(
        train_x,
        train_y_1hot,
        batch_size=batch_size,
        epochs=epochs,
        validation_split=0.2,
        shuffle=False,
    )
    weights = filter_weights(model)
    return history, weights


# plot learning curves
def plot_history(history):
    epochs = len(history.history["accuracy"])
    x = np.arange(1, epochs + 1)
    plt.figure(constrained_layout=True)
    plt.subplot(211)
    plt.plot(x, history.history["accuracy"])
    plt.plot(x, history.history["val_accuracy"], color="green")
    # plt.xlabel("epochs")
    # plt.ylabel("accuracy")
    plt.legend(["train", "validation"], loc="upper left")

    plt.subplot(212)
    plt.plot(x, history.history["loss"])
    plt.plot(x, history.history["val_loss"], color="green")
    plt.xlabel("epochs")
    plt.ylabel("loss")
    plt.legend(["train", "validation"], loc="upper right")
```

In [ ]:
```
# default network for different batch sizes
batches = [1, 256,num_samples_training]
for batch in batches:
    set_seed(1)  # get reproducible results
    print("\n\nBatch size: " + str(batch))
    # default optimizer adam
    model = create_model()
    weights = filter_weights(model)
    plot_weights(weights)

    t = TicToc()
    t.tic()
    history, weight = fitWrapper(batch_size=batch, epochs=100)
    t.toc()

    result = model.evaluate(test_x, test_y_1hot)
    print("Accuracy: " + str(result))
    plot_weights(weight)
    plot_history(history)
```

In [ ]:
```
# rmsprop

rhos = [0.01, 0.99]
for rho in rhos:
    set_seed(1)

    model = create_model(optimizer=RMSprop(learning_rate=0.001, rho=rho))
    weights = filter_weights(model)
    plot_weights(weights)

    history, weights = fitWrapper(batch_size=256, epochs=100)

    weights = filter_weights(model)
```

```
    plot_weights(weights)
    plot_history(history)
    print("Evalute", model.evaluate(test_x, test_y_1hot))
```

In [ ]:
```
# sgd + weight initialization
set_seed(1)

model = create_model(
    optimizer=SGD(lr=0.01),
    kernel_initializer=RandomNormal(mean=10))
weights = filter_weights(model)
plot_weights(weights)

history, weights = fitWrapper(batch_size=256, epochs=100)

weights = filter_weights(model)
plot_weights(weights)
plot_history(history)
print("Evalute", model.evaluate(test_x, test_y_1hot))
```

In [ ]:
```
# l2 regularization model + rmsprop
alphas = [0.1, 0.01, 0.001]
for alpha in alphas:
    set_seed(1)

    model = create_model(
        optimizer=RMSprop(learning_rate=0.001, rho=0.99),
        kernel_regularizer=l2(alpha)
    )
    weights = filter_weights(model)
    plot_weights(weights)

    history, weights = fitWrapper(batch_size=256, epochs=100)

    weights = filter_weights(model)
    plot_weights(weights)
    plot_history(history)
    print("Evaluate", alpha, model.evaluate(test_x, test_y_1hot))
```

In [ ]:
```
# l2 regularization model + sgd
alphas = [0.1, 0.01, 0.001]
for alpha in alphas:
    set_seed(1)

    model = create_model(
        optimizer=SGD(lr=0.01),
        kernel_initializer=RandomNormal(10),
        kernel_regularizer=l2(alpha))
    weights = filter_weights(model)
    plot_weights(weights)

    history, weights = fitWrapper(batch_size=256, epochs=100)

    weights = filter_weights(model)
    plot_weights(weights)
    plot_history(history)
    print("Evaluate", alpha, model.evaluate(test_x, test_y_1hot))
```

In [ ]:
```
# l1-dropout regularization rmsprop
set_seed(1)

model = create_model(
    optimizer=RMSprop(learning_rate=0.001, rho=0.99),
    kernel_regularizer=l1(0.01),
```

```
    is_dropout=True,
)
weights = filter_weights(model)
plot_weights(weights)

history, weights = fitWrapper(batch_size=256, epochs=100)

weights = filter_weights(model)
plot_weights(weights)
plot_history(history)
print("Evalute", model.evaluate(test_x, test_y_1hot))
```

In [ ]:
```
# l1-dropout regularization sgd + initialization
set_seed(1)

model = create_model(
    optimizer=SGD(lr=0.01),
    kernel_initializer=RandomNormal(10),
    kernel_regularizer=l1(0.01),
    is_dropout=True,
)
weights = filter_weights(model)
plot_weights(weights)

history, weights = fitWrapper(batch_size=256, epochs=100)

weights = filter_weights(model)
plot_weights(weights)
plot_history(history)
print("Evalute", model.evaluate(test_x, test_y_1hot))
```

In [ ]:
```
#  Fine-tuning
from keras.callbacks import EarlyStopping

set_seed(1)

# custom metric functions
# source: https://github.com/keras-team/autokeras/issues/867#issuecomment-664794336
from keras import backend as K


def recall_m(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    possible_positives = K.sum(K.round(K.clip(y_true, 0, 1)))
    recall = true_positives / (possible_positives + K.epsilon())
    return recall


def precision_m(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
    precision = true_positives / (predicted_positives + K.epsilon())
    return precision


def f1_score(y_true, y_pred):
    precision = precision_m(y_true, y_pred)
    recall = recall_m(y_true, y_pred)
    return 2 * ((precision * recall) / (precision + recall + K.epsilon()))


def build_model(hp):
    hidden_layer_nodes1 = hp.Choice("hidden_layer_nodes1", values=[64, 128])
    hidden_layer_nodes2 = hp.Choice("hidden_layer_nodes2", values=[256, 512])
    learning_rate = hp.Choice("learning_rate", values=[0.1, 0.01, 0.001])
```

```
    l2_alpha = hp.Choice("l2_alpha", values=[0.1, 0.001, 0.000001])
    return create_model(
        hidden_layer_nodes1=hidden_layer_nodes1,
        hidden_layer_nodes2=hidden_layer_nodes2,
        kernel_regularizer=l2(l2_alpha),
        kernel_initializer=HeNormal(),
        optimizer=RMSprop(learning_rate=learning_rate),
        metrics=["accuracy", f1_score, recall_m, precision_m],
    )


build_model(kt.HyperParameters())
```

Choose your HyperBand

Objective can be:

- `val_f1_score`
- `val_accuracy`

In [ ]:
```
# https://neptune.ai/blog/keras-tuner-tuning-hyperparameters-deep-learning-model
# we could use `val_f1`` ? But we will have bad results?
# `val_f1` isn't supported but you can define cusotm f1 function
# https://github.com/keras-team/autokeras/issues/867


tuner = kt.Hyperband(hypermodel=build_model, objective=Objective("val_f1_score", dire
tuner.search(
    train_x,
    train_y_1hot,
    validation_split=0.2,
    epochs=1000,
    callbacks=[EarlyStopping(patience=200, monitor="val_loss")],
)
tuner.results_summary()
```

In [ ]:
```
tuner = kt.Hyperband(hypermodel=build_model, objective="val_accuracy")
tuner.search(
    train_x,
    train_y_1hot,
    validation_split=0.2,
    epochs=1000,
    callbacks=[EarlyStopping(patience=200, monitor="val_loss")],
)
tuner.results_summary()
```

In [ ]:
```
# train with the best hyperparameters

best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
best_model = tuner.hypermodel.build(best_hps)

history = best_model.fit(train_x, train_y_1hot, epochs=50, validation_split=0.2)
loss_val, accuracy_val, f1_score_val, recall_val, precision_val = best_model.evaluate
    test_x, test_y_1hot
)

print("loss:" + str(loss_val))
print("accuracy:" + str(accuracy_val))
print("f1 score:" + str(f1_score_val))
print("recall:" + str(recall_val))
print("precision:" + str(precision_val))
```

```
# learning curves
plot_history(history)
```

In [ ]:
```
# this one is already trained by getting the best model and we can observe overfittin
# we deduce that by the starting training accuracy of 1.0
# it is better to create the model by the hyperparameters

# train, test the best model
best_model = tuner.get_best_models(num_models=1)[0]
history = best_model.fit(train_x, train_y_1hot, epochs=10, validation_split=0.2)
loss_val, accuracy_val, f1_score_val, recall_val, precision_val = best_model.evaluate
    test_x, test_y_1hot
)

print("loss:" + str(loss_val))
print("accuracy:" + str(accuracy_val))
print("f1 score:" + str(f1_score_val))
print("recall:" + str(recall_val))
print("precision:" + str(precision_val))

# learning curves
plot_history(history)
```

In [ ]:
```
# confusion matrix
from sklearn.metrics import confusion_matrix

y_pred = best_model.predict(test_x)
confusion_mat = confusion_matrix(test_y, y_pred.argmax(axis=1))
normed_conf = (confusion_mat.T / confusion_mat.astype(float).sum(axis=1)).T

fig, ax = plt.subplots(figsize=(10, 10))
sns.heatmap(normed_conf, annot=True, fmt=".2f")
plt.ylabel("Actual")
plt.xlabel("Predicted")
```