Data-Driven Techniques: 1st Homework

Μαρία-Νίκη Ζωγράφου

AM: 1096060

Περιεχόμενα

Ερώτημα 1:

Ερώτημα α:

What is the optimum Bayes test that minimizes the decision error probability?

Το βέλτιστο Bayes Test για να ελαχιστοποιηθεί το κόστος ορίζεται ως:

$$Αν \quad \frac{f_1(x)}{f_0(x)} > \frac{(C_{10}-C_{00})P(H_0)}{(C_{01}-C_{11})P(H_1)} \Rightarrow H_1$$

$$Αν \quad \frac{f_1(x)}{f_0(x)} < \frac{(C_{10}-C_{00})P(H_0)}{(C_{01}-C_{11})P(H_1)} \Rightarrow H_0$$

Στη συγκεκριμένη περίπτωση ορίζουμε $C_{10} = C_{01} = 1$ και $C_{00} = C_{11} = 0$, επομένως ο λόγος πιθανοφάνειας likelihood ratio ορίζεται ως:

$$Αν \quad \frac{f_1(x)}{f_0(x)} > \frac{P(H_0)}{P(H_1)} \Rightarrow H_1$$

$$Αν \quad \frac{f_1(x)}{f_0(x)} < \frac{P(H_0)}{P(H_1)} \Rightarrow H_0$$

Κατανομή της H_0: x_1, x_2 ανεξάρτητες με συνάρτηση πυκνότητας πιθανότητας f0(x1,x2) = f0(x1)*f0(x2) όπου f0(x) ~ N(0,1). Καθώς ακολουθεί την κανονική κατανομή, μπορούμε να υπολογίσουμε την f0 ως:

$$Κανονική\ Κατανομή\ pdf: \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

$$Επομένως\ f_0(x1,x2) = f_0(x1) * f_0(x2) = \left(\frac{1}{\sqrt{2\pi}}\right)^2 \left(e^{-\frac{1}{2}(x_1)^2}\right)\left(e^{-\frac{1}{2}(x_2)^2}\right)$$

Η κατανομή της H_1: x_1, x_2 ανεξάρτητες με συνάρτηση πυκνότητας πιθανότητας f1(x1,x2) = f1(x1)*f1(x2) where f1(x) ~ 0.5{N(−1,1) +N(1,1)}. Επομένως υπάρχει 50% πιθανότητα να ακολουθούν την N(−1,1) και 50% να ακολουθούν την N(1,1). Υπολογίζουμε την f1 ως:

$$f_1(x1,x2) = \left(\frac{0.5}{\sqrt{2\pi}}\right)^2 \left(e^{-\frac{1}{2}(x_1+1)^2} + e^{-\frac{1}{2}(x_1-1)^2}\right)\left(e^{-\frac{1}{2}(x_2+1)^2} + e^{-\frac{1}{2}(x_2-1)^2}\right)$$

Επομένως

$$r(x1,x2) = \frac{f1(x1,x2)}{f0(x1,x2)}$$

$$r(x1,x2) = \frac{\left(\frac{0.5}{\sqrt{2\pi}}\right)^2 \left(e^{-\frac{1}{2}(x_1+1)^2} + e^{-\frac{1}{2}(x_1-1)^2}\right)\left(e^{-\frac{1}{2}(x_2+1)^2} + e^{-\frac{1}{2}(x_2-1)^2}\right)}{\left(\frac{1}{\sqrt{2\pi}}\right)^2 \left(e^{-\frac{1}{2}(x_1)^2}\right)\left(e^{-\frac{1}{2}(x_2)^2}\right)}$$

$$r(x1,x2) = (0.5)^2\left(e^{-\frac{1}{2}(x_1+1)^2+\frac{1}{2}(x_1)^2} + e^{-\frac{1}{2}(x_1-1)^2+\frac{1}{2}(x_1)^2}\right)\left(e^{-\frac{1}{2}(x_2+1)^2+\frac{1}{2}(x_2)^2} + e^{-\frac{1}{2}(x_2-1)^2+\frac{1}{2}(x_2)^2}\right)$$

$$r(x1,x2) = (0.5)^2\left(e^{-x_1-\frac{1}{2}} + e^{x_1-\frac{1}{2}}\right)\left(e^{-x_2-\frac{1}{2}} + e^{x_2-\frac{1}{2}}\right)$$

$$\boldsymbol{r(x1,x2) = (0.5)^2 e^{-1}(e^{-x_1} + e^{x_1})(e^{-x_2} + e^{x_2})}$$

Η εκφώνηση δίνει δεδομένο ότι $P(H_0) = P(H_1) = 0.5$, επομένως $\frac{P(H_0)}{P(H_1)} = 1$.

Άρα αν $\boldsymbol{r(x1,x2)}$ **> 1 επιλέγουμε H1, αν** $\boldsymbol{r(x1,x2)}$ **< 1 επιλέγουμε H0. Αν** $\boldsymbol{r(x1,x2) = 1}$ **δεν μπορούμε να προβλέψουμε.**

Ερώτημα β

Κανόνας του Bayes πάνω σε τυχαία δεδομένα:

Α τρόπος:

- Δημιουργούμε δύο κατηγορίες δειγμάτων (x1,x2)
  - H0: f0 ~ N(0,1)
  - H1: f1 ~ 0.5 * N(-1,1) + 0.5 * N(1,1)
- Αφού δημιουργήσουμε τις δύο κατηγορίες δειγμάτων εφαρμόζουμε τον κανόνα Bayes για κάθε δείγμα της πρώτης κατανομής και μετράμε τα ποσοστά των λανθασμένων αποφάσεων. Επαναλαμβάνουμε τη διαδικασία για την δεύτερη περίπτωση.
- Αθροίζουμε τα λάθη για να βρούμε το τελικό αθροιστικό ποσοστό σφάλματος (error_percentage = errors / total_decisions * 100).

Αποτελέσματα:

```
error percentage: 35.3983%
error percentage for H0: 28.2317%
error percentage for H1: 42.5649%
```

Β τρόπος (πιο ακριβής προσέγγιση):

Υπάρχει και ένας δεύτερος τρόπος υλοποίησης των παραπάνω που λαμβάνει υπόψιν του την πιθανότητα ύπαρξης πολύ μεγάλων ή πολύ μικρών τιμών των x1,x2 που ως εκθέτες μπορεί να δώσουν στις πυκνότητες f0(x) και f1(x) εξαιρετικά μικρές τιμές. Έτσι μπορεί να υπάρξει **overflow ή underflow** και τα τελικά αποτελέσματα να είναι λανθασμένα.

Αυτό το πρόβλημα μπορούμε να το αποφύγουμε υπολογίζοντας τον λογάριθμο του λόγου πιθανοφάνειας. Οι λογάριθμοι μετασχηματίζουν το πρόβλημα από τον πολλαπλασιασμό πυκνοτήτων στην άθροιση λογαρίθμων. Έτσι διασφαλίζουμε αριθμητική σταθερότητα και ακρίβεια:

$$\log\big(r(x_1,x_2)\big) = \log\big(f_1(x_1)\big) + \log\big(f_1(x_2)\big) - \big(\log\big(f_0(x_1)\big) + \log\big(f_0(x_2)\big)\big)$$

Σημείωση: Στην συνάρτησης f1 υπάρχει μίξη κατανομών και άθροιση τους άρα καταλήγει σε μορφή log-sum-exp. Οπότε για να υπολογίσουμε την πιθανότητα χρησιμοποιούμε την τεχνική log-sum-exp. (εξήγηση στο hyperlink)

Αποτελέσματα:

```
Bayes error percentage: 35.2472%
Bayes error percentage for H0: 28.1012%
Bayes error percentage for H1: 42.3932%
```

Γνωρίζουμε ότι αυτά είναι τα βέλτιστα ποσοστά λαθών. Κώδικας και για τις δύο προσεγγίσεις στο τέλος της αναφοράς.

Ερώτημα γ

Εκπαίδευση Νευρωνικών Δικτύων 2x20x1:

Training Data:

Δημιουργούμε 200 νέα τυχαία ζευγάρια από τις κατανομές H0 και H1

Αρχιτεκτονική νευρωνικού δικτύου:

Είσοδος δύο διαστάσεων: x1,x2

Κρυφό επίπεδο με 20 νευρώνες: activation function `tanh`.

Έξοδος: 1 νευρώνας με activation function `sigmoid` για το Cross-Entropy Loss και `γραμμική` για το Exponential Loss.

Κάθε επίπεδο l εκτελεί τους εξής υπολογισμούς:

$$A_l = f(Z_l) = f(W_l A_{l-1} + b_l)$$

Όπου f η activation function και W,b τα βάρη και biases (μεροληψίες).

Stochastic Gradient Descent:

Το νευρωνικό δίκτυο είναι μια παραμετρική συνάρτηση u(x,θ) που με βελτιστοποίηση προσπαθεί να προσεγγίσει τον Bayes. Αντί για σύγκριση του λόγου r(x) με το 1 γίνεται σύγκριση του ω(r(x)) με το ω(1).

Ορίζουμε συνάρτηση κόστους, την οποία θέλουμε να ελαχιστοποιήσουμε:

$$\boldsymbol{J(u(x)) = E_0[\varphi(u(x))] + E_1[\psi(u(x))]}$$

Η βέλτιστη λύση είναι η $u_0(x)$=avgminJ(u). Πρέπει να επιλεχθούν οι κατάλληλες φ και ψ συναρτήσεις έτσι ώστε η βέλτιστη λύση u_0 να είναι ο μετασχηματισμός ω(r(x)). Για το cross-entropy και το exponential έχουν βρεθεί στην παράδοση του μαθήματος οι κατάλληλες συναρτήσεις φ και ψ.

Με την χρήση δεδομένων για training, για κάθε παράμετρο θ (είτε τα βάρη(weights) είτε τα biases b) εκτελείται η εξής επαναληπτική διαδικασία:

$$\theta_t = \theta_{t-1} - \mu\nabla_\theta\{\varphi\left(u(x_{t,0}, \theta_{t-1})\right) + \psi(u(x_{t,1}\theta_{t-1}))\}$$

Αν δεν έχω ζευγάρια δεδομένων, απλά όταν το x προήλθε από H0:

$$\theta_t = \theta_{t-1} - \mu \nabla_\theta \varphi \left(u\left(x_{t,0}, \theta_{t-1}\right)\right)$$

Ενώ αν το x προήλθε από H1:

$$\theta_t = \theta_{t-1} - \mu \nabla_\theta \psi (u(x_{t,1} \theta_{t-1}\,))$$

Adam Optimization:

Ο Adaptive Moment Estimation είναι ένας αλγόριθμος για βελτιστοποίηση του SGD, προκειμένου να επιτευχθεί γρηγορότερα η σύγκλιση.

- Υπολογίζει έναν κινητό μέσο όρο των πρώτων ροπών των παραγώγων (δηλαδή, των παραγώγων των παραμέτρων) για να επιταχύνει την εκπαίδευση.
- Υπολογίζει έναν κινητό μέσο όρο των τετραγώνων των παραγώγων, προσαρμόζοντας το βήμα εκπαίδευσης για κάθε παράμετρο.

Για κάθε παράμετρο θ, που είναι είτε τα βάρη(weights) είτε τα biases b κάνουμε:

1 υπολογισμός στιγμιαίας παραγώγου συνάρτησης κόστους:

$$g_t = \nabla_\theta J(\theta)$$

2 Υπολογισμός κινητών μέσων όρων(λαμβάνουν υπόψιν και τις παλαιότερες τιμές των m_t,u_t):

$$Momentum\colon\ m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$RMSprop\colon\ u_t = \beta_2 u_{t-1} + (1-\beta_2) g_t^2$$

3 Bias Correction:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{u}_t = \frac{u_t}{1-\beta_2^t}$$

4 Ενημέρωση παραμέτρων (*η είναι το learning rate*, και *ε* μια μικρή σταθερά για αποφυγή διαίρεσης με το μηδέν):

$$\theta_t = \theta_{t-1} - \eta\, \frac{\hat{m}_t}{\sqrt{\hat{u}_t} + \varepsilon}$$

Εφαρμόζουμε την παραπάνω μεθοδολογία με τις κατάλληλες ψ και φ για τα δύο νερωνικά δίκτυα.

Χρήση Batches στην Εκπαίδευση:

Τα batches χωρίζουν τα δεδομένα εκπαίδευσης σε μικρότερα υποσύνολα που υπολογίζονται διαδοχικά κατά τη διάρκεια της εκπαίδευσης. Το batch size είναι ο αριθμός δειγμάτων δεδομένων που χρησιμοποιούνται σε κάθε βήμα εκπαίδευσης. Στο SGD batch_size=1. Για επιτάχυνση της διαδικασίας και μείωση του θορύβου υπολογίζουμε τα gradients για ένα πλήθος δεδομένων και όχι για ένα δείγμα κάθε φορά. Το gradient δηλαδή είναι μέσος όρος πολλών δειγμάτων.

Τα βάρη $W1$ πολλαπλασιάζονται με όλα τα δείγματα του batch. Η μεροληψία $b1$ προστίθεται σε κάθε δείγμα. Το αποτέλεσμα $Z1$ περιέχει τον γραμμικό συνδυασμό εισόδων για κάθε δείγμα του batch, επιταχύνοντας την εκπαίδευση μέσω παραλληλίας.

Αποτέλεσμα κάθε επιπέδου:

[z11+b1 z12+b1 z13+b1 z14+b1

z21+b2 z22+b2 z23+b2 z24+b2

z31+b3 z32+b3 z33+b3 z34+b3]

Η συνάρτηση κόστους J θα διαιρείται με το μέγεθος του Batch (cost = -np.sum(y_batch * np.log(A2 + epsilon) + (1 - y_batch) * np.log(1 - A2 + epsilon)) / m). Ύστερα θα υπολογιστούν τα gt που αντιστοιχούν στις παραγώγους του κόστους J σε σχέση με τα βάρη και τις μεροληψίες του νευρωνικού δικτύου. Αυτές οι παράγωγοι υπολογίζονται μέσω του backpropagation.

Cross-Entropy Network

$$\omega(r) = \frac{r}{r+1} \in [0,1], posterior\ probability$$

$$\rho(z) = -\frac{1}{z}, z \in [0,1] \Rightarrow \varphi(z) = -log(1-z), \psi(z) = -\log(z)$$

Στην συγκεκριμένη περίπτωση ω(1)=0.5 (ω γνησίως αύξουσα).

1. Αρχικοποιούμε τα βάρη: Δίνουμε στα biases τιμή 0 και στα weights τιμές από μια κανονική κατανομή N(0, sqrt(2 / (n + m))). Το m είναι ο αριθμός των εξόδων του επιπέδου, δηλαδή ο αριθμός νευρώνων στο τρέχον επίπεδο και το n ο αριθμός των εισόδων του επιπέδου, δηλαδή ο αριθμός νευρώνων στο προηγούμενο επίπεδο. Η κατανομή θέλουμε να μας επιστρέψει m γραμμές και n στήλες.
2. Ορίζουμε τις πράξεις που εκτελεί κάθε επίπεδο

Κάθε επίπεδο l εκτελεί τους εξής υπολογισμούς:

$$A_l = f(Z_l) = f(W_l A_{l-1} + b_l)$$

Όπου f η activation function και W,b τα βάρη και biases.

Το κρυφό επίπεδο έχει: $\boldsymbol{f = \tanh(z)}$ και η έξοδος έχει sigmoid $\boldsymbol{\sigma(z) = \frac{1}{1+e^{-z}}}$, η οποία περιορίζει την έξοδο του νευρώνα ανάμεσα σε 0 και 1.

3. Αρχικοποιούμε τις τιμές για τον ADAM optimizer και τα training parameters:

Adam: learning_rate = 0.001, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8

Training parameters: max_epochs = 10000, batch_size = 32

4. Εκτελούμε την εκπαίδευση με τις τεχνικές που αναλύσαμε παραπάνω.
5. Προβλέψεις για τα testing δεδομένα (τα αρχικά 10^6 ζεύγη)

Στα ***predictions*** όταν η τελική έξοδος είναι >= 0.5, επιλέγουμε H1, αλλιώς H0. Συγκρίνουμε την επιλογή μας με το label και μετράμε error percentages.

Exponential Network

$$\omega(r) = log(r) \in \mathbb{R}, \log likelihood\ ratio$$

$$\rho(z) = e^{-0.5z} \Rightarrow \varphi(z) = 2e^{0.5z}, \psi(z) = 2e^{-0.5z}$$

Στην συγκεκριμένη περίπτωση ω(1)=log(1)=0 (ω γνησίως αύξουσα).

1. Αρχικοποιούμε τα βάρη: Δίνουμε στα biases τιμή 0 και στα weights τιμές από μια κανονική κατανομή N(0, sqrt(2 / (n + m))). Το m είναι ο αριθμός των εξόδων του επιπέδου, δηλαδή ο αριθμός νευρώνων στο τρέχον επίπεδο και το n ο αριθμός των εισόδων του επιπέδου, δηλαδή ο αριθμός νευρώνων στο προηγούμενο επίπεδο. Η κατανομή θέλουμε να μας επιστρέψει m γραμμές και n στήλες.
2. Ορίζουμε τις πράξεις που εκτελεί κάθε επίπεδο

Κάθε επίπεδο l εκτελεί τους εξής υπολογισμούς:

$$A_l = f(Z_l) = f(W_l A_{l-1} + b_l)$$

Όπου f η activation function και W,b τα βάρη και biases.

Το κρυφό επίπεδο έχει: $\boldsymbol{f = \tanh(z)} \in [-\mathbf{1}, \mathbf{1}]$ και η έξοδος έχει τη γραμμική συνάρτηση Αεξοδου=Ζεξόδου, η οποία δεν περιορίζει τις τελικές τιμές ανάμεσα σε 0 και 1.

3. Αλλαγή των labels από (0,1) σε (-1,1):

Η συνάρτηση κόστους βασίζεται κάθε φορά σε έναν εκθετικό όρο ($\varphi(z) = 2e^{0.5z}$, $\psi(z) = 2e^{-0.5z}$), οπότε αν το z=0 θα επέστρεφε πάντα τιμή 1. Επομένως χρησιμοποιούμε για labels: {-1 και 1}.

4. Αρχικοποιούμε τις τιμές για τον ADAM optimizer και τα training parameters:

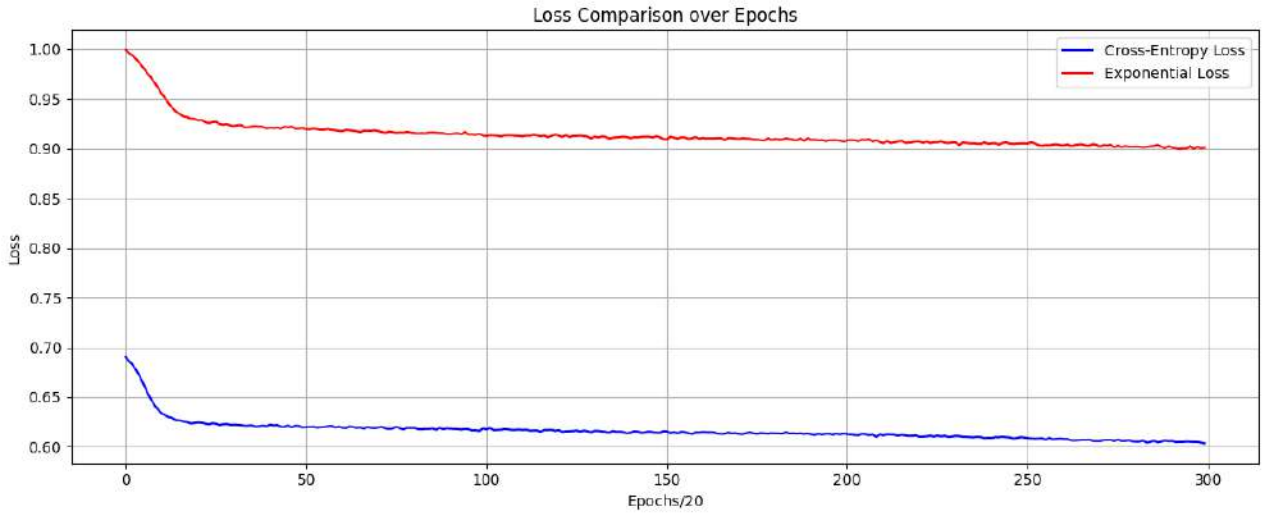
Adam: learning_rate = 0.001, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8

Training parameters: max_epochs = 10000, batch_size = 32

5. Εκτελούμε την εκπαίδευση με τις τεχνικές που αναλύσαμε παραπάνω.
6. Προβλέψεις για τα testing δεδομένα (τα αρχικά 10^6 ζεύγη)

Οι προβλέψεις γίνονται απευθείας από την έξοδο. Αν η έξοδος>=0 επιλέγουμε Η1, αλλιώς Η0.

Σύγκλιση νευρωνικών δικτύων:



Σημείωση: Για καλύτερη γραφική έχει σχεδιαστεί η smoothed loss function τους κάθε δικτύου, η οποία βρίσκει τον μέσο όρο ανά 20 εποχές, ώστε να μην εμφανίζονται τόσα spikes στο γράφημα.

Παρατήρηση: Τα δυο νευρωνικά δίκτυα φαίνεται να συγκλίνουν με την ίδια ταχύτητα. Το Loss του cross entropy τείνει στο 0.6 ενώ του exponential στο 0.9.

Αποτελέσματα:

=== Error Rates Comparison ===

Bayes Optimal Error Percentage: 35.33505%

Bayes error percentage for H0: 28.233000000000004%

Bayes error percentage for H1: 42.4371%

Cross-Entropy Network Error Percentage: 37.1519%

Error Percentage for H0: 30.2115%

Error Percentage for H1: 44.0923%

Exponential Loss Network Error Percentage: 37.7549%

Error Percentage for H0: 32.567800000000005%

Error Percentage for H1: 42.942%

Παρατήρηση: Τα δυο νευρωνικά προσεγγίζουν πολύ καλά τον Bayes, έχοντας σφάλμα λίγο περισσότερο από το βέλτιστο.

Ερώτημα 2

Χρησιμοποιώντας όλες τις παραπάνω τεχνικές για την εκπαίδευση των νευρωνικών δικτύων θα χρησιμοποιήσουμε ως δεδομένα εικόνες «8» και «1» από το MNIST dataset. Η MNIST έχει 5500 εικόνες διαστάσεων 28 × 28 σε αποχρώσεις του γκρι για training.

1. Προετοιμασία δεδομένων:

Φορτώνουμε τις εικόνες και τις χωρίζουμε σε training και testing data. Διαιρούμε με το 255 τις τιμές του γκρι για να τις μεταφέρουμε σε ένα εύρος μεταξύ 0 και 1. Για το cross entropy χρησιμοποιούμε labels 0 και 1 και για το exponential labels -1 και 1.

2. Αρχιτεκτονική των νευρωνικών δικτύων:

Χρησιμοποιούμε νευρωνικά δίκτυα διαστάσεων 784×300×1.

a. Αρχιτεκτονική του Cross Entropy:
   Το κρυφό επίπεδο χρησιμοποιεί για activation function την ReLu (Εφαρμόζεται στον γραμμικό συνδυασμό των εισόδων για να εισάγει μη γραμμικότητα, f(x)=maximum(0, x), δηλαδή οι αρνητικές τιμές γίνονται 0).

   Το τελευταίο επίπεδο χρησιμοποιεί ξανά την sigmoid activation function, η οποία μεταφέρει τις εξόδους στο διάστημα (0,1).

b. Αρχιτεκτονική του Exponential:

Το κρυφό επίπεδο χρησιμοποιεί την tanh(z) για activation function.

Το τελευταίο επίπεδο χρησιμοποιεί ξανά την γραμμική activation function.

3. Εκπαίδευση και Testing:

Χρησιμοποιούμε τις ίδιες τεχνικές με το ερώτημα 1: SGD, Adam Optimizer, Batch size=32 και συναρτήσεις κόστους J, που παραμένουν ίδιες με πριν. Χρησιμοποιούμε όμως διαφορετικό αριθμό εποχών **epochs=50**. Στο cross entropy συγκρίνουμε ξανά με 0.5 και στο exponential με 0.

Αποτελέσματα:

############### CROSS-ENTROPY NETWORK ###################

Cross-Entropy Network Error: 0.46%

error_percentage for 0: 0.40816326530612246%

error_percentage for 8: 0.5133470225872689%

total_error_percentage: 0.46059365404298874%

################## EXPONENTIAL ################

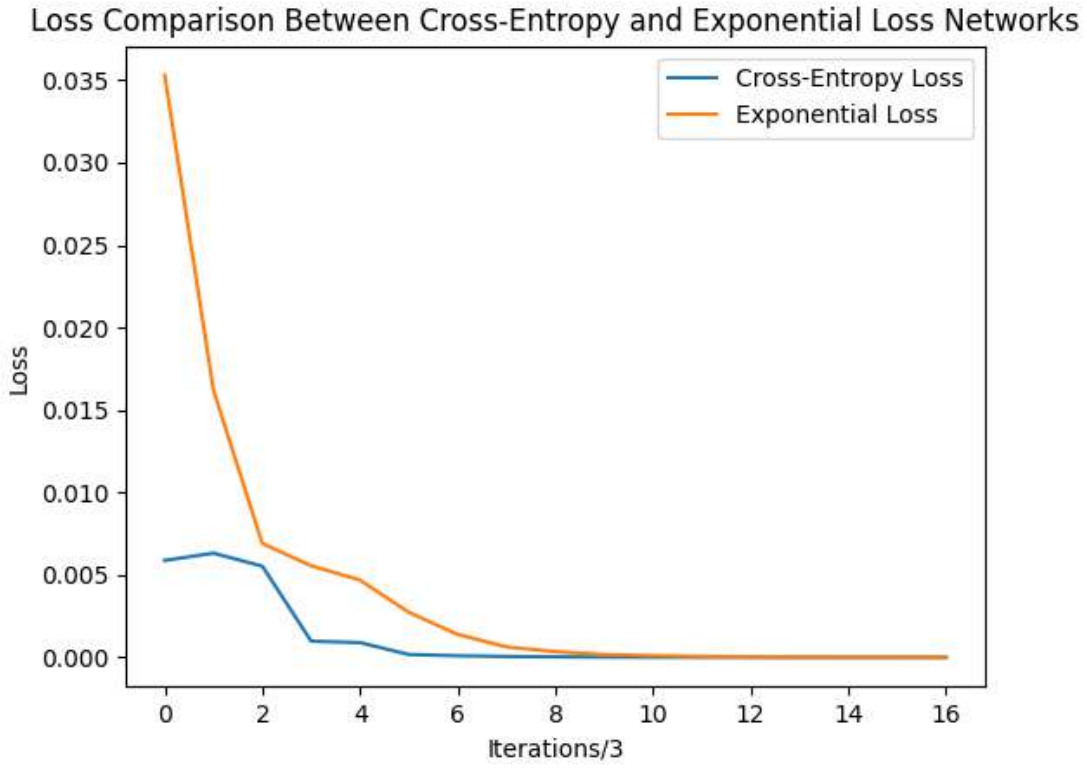
Exponential Network Error: 0.41%

error_percentage for 0: 0.30612244897959184%

error_percentage for 8: 0.5133470225872689%

total_error_percentage: 0.4094165813715456%

Σύγκλιση νευρωνικών δικτύων:



Σημείωση: Για καλύτερη γραφική έχει σχεδιαστεί η smoothed loss function τους κάθε δικτύου, η οποία βρίσκει τον μέσο όρο ανά 3 εποχές, ώστε να μην εμφανίζονται τόσα spikes στο γράφημα.

Παρατηρήσεις: Τα δυο νευρωνικά δίκτυα έχουν ικανοποιητική ακρίβεια με σφάλμα μόλις 0.41% το exponential και 0.46% το cross entropy. Το exponential παρουσιάζει ελαφρά μικρότερο σφάλμα στην ανίχνευση των 0. Και οι δυο αλγόριθμοι συγκλίνουν πριν τις 50 εποχές. **Το cross entropy δίκτυο φαίνεται να συγκλίνει γρηγορότερα από το exponential.**

Κώδικας

Ερώτημα 1β

Α τρόπος

```
import numpy as np
######################      DATA GENERATION      ######################
n_samples = 10**6
######## H0 f0~N(0,1)
mu, sigma = 0, 1 # mean and standard deviation
xH0 = np.random.normal(mu, sigma, (n_samples, 2))
######## H1 f1 ~ 0.5 * ( N(-1,1) + N(1,1) )
xH1=np.where(np.random.rand(n_samples, 1)<0.5,np.random.normal(-1, 1,
(n_samples, 2)),np.random.normal(1, 1, (n_samples, 2)))
######################      Bayes Rule      ######################
# f0(x) = N(0, 1) density
def f0(x) :
    return np.exp(-x**2 / 2) / np.sqrt(2 * np.pi)
# f1(x) = 0.5 * N(-1,1) + 0.5 * N(1,1) density
def f1(x):
    return 0.5 * (np.exp(-(x + 1)**2 / 2) + np.exp(-(x - 1)**2 / 2)) /
np.sqrt(2 * np.pi)
errors=0
err_0=0
err_1=0
for sample in xH0:
    q=(f1(sample[0])*f1(sample[1]))/(f0(sample[0])*f0(sample[1]))
    if q>1: err_0+=1
for sample in xH1:
    q=(f1(sample[0])*f1(sample[1]))/(f0(sample[0])*f0(sample[1]))
    if q<1: err_1+=1
errors=err_0+err_1
total_decisions=n_samples*2
error_percentage=float(errors)/float(total_decisions)
print(f"error percentage: {error_percentage*100}%")
print(f"error percentage for H0:
{(float(err_0)/float(n_samples))*100}%")
print(f"error percentage for H1:
{(float(err_1)/float(n_samples))*100}%")
```

Β τρόπος

```
######################      DATA GENERATION      ######################
n_samples = 10**6
# H0: f0 ~ N(0,1)
xH0 = np.random.normal(0, 1, (n_samples, 2))
# H1: f1 ~ 0.5 * N(-1,1) + 0.5 * N(1,1)
rand_vals = np.random.rand(n_samples, 2)
neg_indices = rand_vals < 0.5
pos_indices = ~neg_indices
xH1 = np.zeros((n_samples, 2))
xH1[neg_indices] = np.random.normal(-1, 1, np.sum(neg_indices))
xH1[pos_indices] = np.random.normal(1, 1, np.sum(pos_indices))
######################      Bayes Rule      ######################
# Density functions
def f0(x):
    return np.exp(-x**2 / 2) / np.sqrt(2 * np.pi)
def f1(x):
    return 0.5 * (np.exp(-(x + 1)**2 / 2) + np.exp(-(x - 1)**2 / 2)) /
np.sqrt(2 * np.pi)
# Compute log-likelihood ratios for numerical stability
def log_f0(x):
    return -0.5 * x**2 - 0.5 * np.log(2 * np.pi)
def log_f1(x):
    log_f1_neg = -0.5 * (x + 1)**2 - 0.5 * np.log(2 * np.pi)
    log_f1_pos = -0.5 * (x - 1)**2 - 0.5 * np.log(2 * np.pi)
    # Sum the probabilities in log-space
    max_log = np.maximum(log_f1_neg, log_f1_pos)
    log_sum = max_log + np.log(0.5 * np.exp(log_f1_neg - max_log) + 0.5
* np.exp(log_f1_pos - max_log))
    return log_sum
# Compute errors under H0
log_q_H0 = np.sum(log_f1(xH0) - log_f0(xH0), axis=1)
err_0 = np.sum(log_q_H0 > 0)
# Compute errors under H1
log_q_H1 = np.sum(log_f1(xH1) - log_f0(xH1), axis=1)
err_1 = np.sum(log_q_H1 <= 0)
# Total error
errors = err_0 + err_1
total_decisions = n_samples * 2
error_percentage = errors / total_decisions * 100
error_percentage_H0 = err_0 / n_samples * 100
error_percentage_H1 = err_1 / n_samples * 100
print(f"Bayes error percentage: {error_percentage}%")
print(f"Bayes error percentage for H0: {error_percentage_H0}%")
print(f"Bayes error percentage for H1: {error_percentage_H1}%")
```

Ερώτημα 1γ (όλο το πρόβλημα 1)

```
import numpy as np
import matplotlib.pyplot as plt

######################      DATA GENERATION      ######################
n_samples = 10**6
# H0: f0 ~ N(0,1)
xH0 = np.random.normal(0, 1, (n_samples, 2))
# H1: f1 ~ 0.5 * N(-1,1) + 0.5 * N(1,1)
rand_vals = np.random.rand(n_samples, 2)
neg_indices = rand_vals < 0.5
pos_indices = ~neg_indices
xH1 = np.zeros((n_samples, 2))
xH1[neg_indices] = np.random.normal(-1, 1, np.sum(neg_indices))
xH1[pos_indices] = np.random.normal(1, 1, np.sum(pos_indices))
######################      Bayes Rule      ######################
# Density functions
def f0(x):
    return np.exp(-x**2 / 2) / np.sqrt(2 * np.pi)
def f1(x):
    return 0.5 * (np.exp(-(x + 1)**2 / 2) + np.exp(-(x - 1)**2 / 2)) /
np.sqrt(2 * np.pi)
# Compute log-likelihood ratios for numerical stability
def log_f0(x):
    return -0.5 * x**2 - 0.5 * np.log(2 * np.pi)
def log_f1(x):
    log_f1_neg = -0.5 * (x + 1)**2 - 0.5 * np.log(2 * np.pi)
    log_f1_pos = -0.5 * (x - 1)**2 - 0.5 * np.log(2 * np.pi)
    # Sum the probabilities in log-space
    max_log = np.maximum(log_f1_neg, log_f1_pos)
    log_sum = max_log + np.log(0.5 * np.exp(log_f1_neg - max_log) + 0.5
* np.exp(log_f1_pos - max_log))
    return log_sum
# Compute errors under H0
log_q_H0 = np.sum(log_f1(xH0) - log_f0(xH0), axis=1)
err_0 = np.sum(log_q_H0 > 0)
# Compute errors under H1
log_q_H1 = np.sum(log_f1(xH1) - log_f0(xH1), axis=1)
err_1 = np.sum(log_q_H1 <= 0)
# Total error
errors = err_0 + err_1
total_decisions = n_samples * 2
error_percentage = errors / total_decisions * 100
error_percentage_H0 = err_0 / n_samples * 100
error_percentage_H1 = err_1 / n_samples * 100
print(f"Bayes error percentage: {error_percentage}%")
print(f"Bayes error percentage for H0: {error_percentage_H0}%")
print(f"Bayes error percentage for H1: {error_percentage_H1}%")
```

```
#####################      Neural Network Training
# Training data: 200 samples from each class
N_train = 200
# Generate training data for H0
xH0_train = np.random.normal(0, 1, (N_train, 2))
yH0_train = np.zeros(N_train)
# Generate training data for H1
rand_vals_train = np.random.rand(N_train, 2)
neg_indices_train = rand_vals_train < 0.5
pos_indices_train = ~neg_indices_train
xH1_train = np.zeros((N_train, 2))
xH1_train[neg_indices_train] = np.random.normal(-1, 1,
np.sum(neg_indices_train))
xH1_train[pos_indices_train] = np.random.normal(1, 1,
np.sum(pos_indices_train))
yH1_train = np.ones(N_train)

# Combine training data
X_train = np.vstack((xH0_train, xH1_train))
y_train = np.hstack((yH0_train, yH1_train))

# Shuffle training data
perm = np.random.permutation(len(X_train))
X_train = X_train[perm]
y_train = y_train[perm]
'''''''
# Normalize input data
mean = np.mean(X_train, axis=0)
std = np.std(X_train, axis=0)
X_train_normalized = (X_train - mean) / std

# Neural network parameters
input_size = 2
hidden_size = 20
output_size = 1

def initialize_weights(m, n):
    std_dev = np.sqrt(2 / (n + m))
    weights = np.random.normal(0, std_dev, (m, n))
    biases = np.zeros((m, 1))
    return weights, biases

# Initialize weights for Cross-Entropy network
W1_ce, b1_ce = initialize_weights(hidden_size, input_size)
W2_ce, b2_ce = initialize_weights(output_size, hidden_size)

# Initialize weights for Exponential Loss network
```

```
W1_exp, b1_exp = initialize_weights(hidden_size, input_size)
W2_exp, b2_exp = initialize_weights(output_size, hidden_size)

# Training parameters
learning_rate = 0.001
beta1 = 0.9
beta2 = 0.999
epsilon = 1e-8
max_epochs = 10000
batch_size = 32

# Activation functions
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def tanh_derivative(a):
    return 1 - a**2

#################### Cross-Entropy Loss Network Training
# Adam optimizer variables
t_ce = 0
m_W1_ce = np.zeros_like(W1_ce)
v_W1_ce = np.zeros_like(W1_ce)
m_b1_ce = np.zeros_like(b1_ce)
v_b1_ce = np.zeros_like(b1_ce)
m_W2_ce = np.zeros_like(W2_ce)
v_W2_ce = np.zeros_like(W2_ce)
m_b2_ce = np.zeros_like(b2_ce)
v_b2_ce = np.zeros_like(b2_ce)

cost_history_ce = []
smooth_cost_hist=[]
#cost_history = []
smooth_cost_window = 20  # Using a 20-cost window for smoothing
#convergence_threshold = 1e-6  # Define a small threshold for
convergence
for epoch in range(max_epochs):
    # Shuffle training data at the beginning of each epoch
    perm = np.random.permutation(len(X_train))
    X_train = X_train[perm]
    y_train = y_train[perm]
    X_train_normalized = X_train_normalized[perm]

    epoch_cost = 0
    num_batches = int(np.ceil(len(X_train) / batch_size))

    for batch in range(num_batches):
        start = batch * batch_size
```

```
        end = min(start + batch_size, len(X_train))
        X_batch = X_train_normalized[start:end].T
        y_batch = y_train[start:end].reshape(1, -1)

        # Forward pass
        Z1 = np.dot(W1_ce, X_batch) + b1_ce
        A1 = np.tanh(Z1)
        Z2 = np.dot(W2_ce, A1) + b2_ce
        A2 = sigmoid(Z2)
                # Compute cost
        m = y_batch.shape[1]
        cost = -np.sum(y_batch * np.log(A2 + epsilon) + (1 - y_batch) *
np.log(1 - A2 + epsilon)) / m
        epoch_cost += cost

        # Backward pass
        dZ2 = A2 - y_batch
        dW2 = np.dot(dZ2, A1.T) / m
        db2 = np.sum(dZ2, axis=1, keepdims=True) / m
        dA1 = np.dot(W2_ce.T, dZ2)
        dZ1 = dA1 * tanh_derivative(A1)
        dW1 = np.dot(dZ1, X_batch.T) / m
        db1 = np.sum(dZ1, axis=1, keepdims=True) / m

        # Adam optimizer update
        t_ce += 1
        # Update first moment estimates
        m_W1_ce = beta1 * m_W1_ce + (1 - beta1) * dW1
        m_b1_ce = beta1 * m_b1_ce + (1 - beta1) * db1
        m_W2_ce = beta1 * m_W2_ce + (1 - beta1) * dW2
        m_b2_ce = beta1 * m_b2_ce + (1 - beta1) * db2
        # Update second moment estimates
        v_W1_ce = beta2 * v_W1_ce + (1 - beta2) * (dW1 ** 2)
        v_b1_ce = beta2 * v_b1_ce + (1 - beta2) * (db1 ** 2)
        v_W2_ce = beta2 * v_W2_ce + (1 - beta2) * (dW2 ** 2)
        v_b2_ce = beta2 * v_b2_ce + (1 - beta2) * (db2 ** 2)
        # Compute bias-corrected moment estimates
        m_W1_ce_hat = m_W1_ce / (1 - beta1 ** t_ce)
        m_b1_ce_hat = m_b1_ce / (1 - beta1 ** t_ce)
        v_W1_ce_hat = v_W1_ce / (1 - beta2 ** t_ce)
        v_b1_ce_hat = v_b1_ce / (1 - beta2 ** t_ce)
        m_W2_ce_hat = m_W2_ce / (1 - beta1 ** t_ce)
        m_b2_ce_hat = m_b2_ce / (1 - beta1 ** t_ce)
        v_W2_ce_hat = v_W2_ce / (1 - beta2 ** t_ce)
        v_b2_ce_hat = v_b2_ce / (1 - beta2 ** t_ce)
        # Update parameters
        W1_ce -= learning_rate * m_W1_ce_hat / (np.sqrt(v_W1_ce_hat) +
epsilon)
```

```
        b1_ce -= learning_rate * m_b1_ce_hat / (np.sqrt(v_b1_ce_hat) +
epsilon)
        W2_ce -= learning_rate * m_W2_ce_hat / (np.sqrt(v_W2_ce_hat) +
epsilon)
        b2_ce -= learning_rate * m_b2_ce_hat / (np.sqrt(v_b2_ce_hat) +
epsilon)

    cost_history_ce.append(epoch_cost / num_batches)
     # Smoothing cost calculation
    if (len(cost_history_ce) % smooth_cost_window) == 0:
        smoothed_cost = np.mean(cost_history_ce[-smooth_cost_window:])
        smooth_cost_hist.append(smoothed_cost)
        if len(cost_history_ce) > 2*smooth_cost_window:
            cost_dif=np.abs(smooth_cost_hist[-1]-smooth_cost_hist[-2])
            #if cost_dif<convergence_threshold:
            #    print(f"cross entropy Converged at epoch {epoch}")
            #    break
    else:
        smoothed_cost = np.mean(cost_history_ce)
    # Print cost every 1000 epochs
    if (epoch + 1) % 1000 == 0:
        print(f"Epoch {epoch + 1}, Cross-Entropy Loss:
{cost_history_ce[-1]}")

#################### Exponential Loss Network Training
# Adam optimizer variables
t_exp = 0
m_W1_exp = np.zeros_like(W1_exp)
v_W1_exp = np.zeros_like(W1_exp)
m_b1_exp = np.zeros_like(b1_exp)
v_b1_exp = np.zeros_like(b1_exp)
m_W2_exp = np.zeros_like(W2_exp)
v_W2_exp = np.zeros_like(W2_exp)
m_b2_exp = np.zeros_like(b2_exp)
v_b2_exp = np.zeros_like(b2_exp)

cost_history_exp = []
smooth_cost_histexp=[]

# Convert labels to {-1, 1}
y_train_exp = np.where(y_train == 1, 1, -1)

for epoch in range(max_epochs):
    # Shuffle training data at the beginning of each epoch
    perm = np.random.permutation(len(X_train))
    X_train = X_train[perm]
    y_train_exp = y_train_exp[perm]
    X_train_normalized = X_train_normalized[perm]
```

```
    epoch_cost = 0
    num_batches = int(np.ceil(len(X_train) / batch_size))

    for batch in range(num_batches):
        start = batch * batch_size
        end = min(start + batch_size, len(X_train))
        X_batch = X_train_normalized[start:end].T
        y_batch = y_train_exp[start:end].reshape(1, -1)

        # Forward pass
        Z1 = np.dot(W1_exp, X_batch) + b1_exp
        A1 = np.tanh(Z1)
        Z2 = np.dot(W2_exp, A1) + b2_exp
        A2 = Z2  # No activation in output layer

        # Compute cost
        m = y_batch.shape[1]
        cost = np.sum(np.exp(-0.5 * y_batch * A2)) / m
        epoch_cost += cost

        # Backward pass
        dZ2 = -0.5 * y_batch * np.exp(-0.5 * y_batch * A2)
        dW2 = np.dot(dZ2, A1.T) / m
        db2 = np.sum(dZ2, axis=1, keepdims=True) / m
        dA1 = np.dot(W2_exp.T, dZ2)
        dZ1 = dA1 * tanh_derivative(A1)
        dW1 = np.dot(dZ1, X_batch.T) / m
        db1 = np.sum(dZ1, axis=1, keepdims=True) / m

        # Adam optimizer update
        t_exp += 1
        # Update first moment estimates
        m_W1_exp = beta1 * m_W1_exp + (1 - beta1) * dW1
        m_b1_exp = beta1 * m_b1_exp + (1 - beta1) * db1
        m_W2_exp = beta1 * m_W2_exp + (1 - beta1) * dW2
        m_b2_exp = beta1 * m_b2_exp + (1 - beta1) * db2
        # Update second moment estimates
        v_W1_exp = beta2 * v_W1_exp + (1 - beta2) * (dW1 ** 2)
        v_b1_exp = beta2 * v_b1_exp + (1 - beta2) * (db1 ** 2)
        v_W2_exp = beta2 * v_W2_exp + (1 - beta2) * (dW2 ** 2)
        v_b2_exp = beta2 * v_b2_exp + (1 - beta2) * (db2 ** 2)
        # Compute bias-corrected moment estimates
        m_W1_exp_hat = m_W1_exp / (1 - beta1 ** t_exp)
        m_b1_exp_hat = m_b1_exp / (1 - beta1 ** t_exp)
        v_W1_exp_hat = v_W1_exp / (1 - beta2 ** t_exp)
        v_b1_exp_hat = v_b1_exp / (1 - beta2 ** t_exp)
        m_W2_exp_hat = m_W2_exp / (1 - beta1 ** t_exp)
```

```
        m_b2_exp_hat = m_b2_exp / (1 - beta1 ** t_exp)
        v_W2_exp_hat = v_W2_exp / (1 - beta2 ** t_exp)
        v_b2_exp_hat = v_b2_exp / (1 - beta2 ** t_exp)
        # Update parameters
        W1_exp -= learning_rate * m_W1_exp_hat / (np.sqrt(v_W1_exp_hat)
+ epsilon)
        b1_exp -= learning_rate * m_b1_exp_hat / (np.sqrt(v_b1_exp_hat)
+ epsilon)
        W2_exp -= learning_rate * m_W2_exp_hat / (np.sqrt(v_W2_exp_hat)
+ epsilon)
        b2_exp -= learning_rate * m_b2_exp_hat / (np.sqrt(v_b2_exp_hat)
+ epsilon)

    cost_history_exp.append(epoch_cost / num_batches)
     # Smoothing cost calculation
    if (len(cost_history_exp) % smooth_cost_window) == 0:
        smoothed_cost = np.mean(cost_history_exp[-smooth_cost_window:])
        smooth_cost_histexp.append(smoothed_cost)
        if len(cost_history_exp) > 2*smooth_cost_window:
            cost_dif=np.abs(smooth_cost_histexp[-1]-
smooth_cost_histexp[-2])
            #if cost_dif<convergence_threshold:
            #    print(f"exponential Converged at epoch {epoch}")
            #    break
    else:
        smoothed_cost = np.mean(cost_history_exp)
    # Print cost every 1000 epochs
    if (epoch + 1) % 1000 == 0:
        print(f"Epoch {epoch + 1}, Exponential Loss:
{cost_history_exp[-1]}")
###################### Testing on Original Data ######################
# Combine test data
X_test = np.vstack((xH0, xH1))
y_test = np.hstack((np.zeros(n_samples), np.ones(n_samples)))
# Normalize test data using training mean and std
X_test_normalized = (X_test - mean) / std
# Predict with Cross-Entropy network
def predict_ce(X):
    Z1 = np.dot(W1_ce, X.T) + b1_ce
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2_ce, A1) + b2_ce
    A2 = sigmoid(Z2)
    return A2.flatten()
predictions_ce = predict_ce(X_test_normalized)
decisions_ce = np.where(predictions_ce >= 0.5, 1, 0)
errors_ce = np.sum(decisions_ce != y_test)
error_percentage_ce = errors_ce / (2 * n_samples) * 100
```

```
error_percentage_ce_H0 = np.sum(decisions_ce[:n_samples] !=
y_test[:n_samples]) / n_samples * 100
error_percentage_ce_H1 = np.sum(decisions_ce[n_samples:] !=
y_test[n_samples:]) / n_samples * 100
# Predict with Exponential Loss network
def predict_exp(X):
    Z1 = np.dot(W1_exp, X.T) + b1_exp
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2_exp, A1) + b2_exp
    return Z2.flatten()
predictions_exp = predict_exp(X_test_normalized)
decisions_exp = np.where(predictions_exp >= 0, 1, 0)
errors_exp = np.sum(decisions_exp != y_test)
error_percentage_exp = errors_exp / (2 * n_samples) * 100
error_percentage_exp_H0 = np.sum(decisions_exp[:n_samples] !=
y_test[:n_samples]) / n_samples * 100
error_percentage_exp_H1 = np.sum(decisions_exp[n_samples:] !=
y_test[n_samples:]) / n_samples * 100
######################  Comparison of Error Rates ######################
print("\n=== Error Rates Comparison ===")
print(f"Bayes Optimal Error Percentage: {error_percentage}%")
print(f"    Bayes error percentage for H0: {error_percentage_H0}%")
print(f"    Bayes error percentage for H1: {error_percentage_H1}%")
print(f"Cross-Entropy Network Error Percentage:
{error_percentage_ce}%")
print(f"   Error Percentage for H0: {error_percentage_ce_H0}%")
print(f"   Error Percentage for H1: {error_percentage_ce_H1}%")
print(f"Exponential Loss Network Error Percentage:
{error_percentage_exp}%")
print(f"   Error Percentage for H0: {error_percentage_exp_H0}%")
print(f"   Error Percentage for H1: {error_percentage_exp_H1}%")
#################### Plot Training Loss ####################
plt.figure(figsize=(12, 5))
# Plot Cross-Entropy and Exponential Loss on the same plot
plt.plot(smooth_cost_hist, color='blue', label='Cross-Entropy Loss')
plt.plot(smooth_cost_histexp, color='red', label='Exponential Loss')
plt.xlabel('Epochs/20')
plt.ylabel('Loss')
plt.title('Loss Comparison over Epochs')
plt.legend()  # Display legend to differentiate between the two losses
plt.grid(True)
plt.tight_layout()
plt.show()
```

Ερώτημα 2

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
import sys
sys.stdout = open(sys.stdout.fileno(), mode='w', encoding='utf-8',
buffering=1)

def load_mnist():
    (train_images, train_labels), (test_images, test_labels) =
mnist.load_data()
    return train_images, train_labels, test_images, test_labels

train_images, train_labels, test_images, test_labels = load_mnist()

# Filter only the numerals 0 and 8
def filter_data(images, labels, num1=0, num2=8):
    filter_mask = (labels == num1) | (labels == num2)
    return images[filter_mask], labels[filter_mask]

train_images, train_labels = filter_data(train_images, train_labels)
test_images, test_labels = filter_data(test_images, test_labels)

# Convert labels to binary classification (0 or 1)
train_labels = (train_labels == 8).astype(int)
test_labels = (test_labels == 8).astype(int)

# Normalize images to the range [0, 1]
train_images = train_images / 255.0
test_images = test_images / 255.0

# Flatten images to vectors of length 784
train_images = train_images.reshape(train_images.shape[0], -1)
test_images = test_images.reshape(test_images.shape[0], -1)

# Initialize neural network parameters for Cross-Entropy Network with
Adam Optimizer
np.random.seed(0)
input_size = 784
hidden_size = 300
output_size = 1

# Weights initialization for Cross-Entropy Network
w1_ce = np.random.normal(0, np.sqrt(1 / (input_size + hidden_size)),
(input_size, hidden_size))
b1_ce = np.zeros(hidden_size)
w2_ce = np.random.normal(0, np.sqrt(1 / (hidden_size + output_size)),
(hidden_size, output_size))
```

```
b2_ce = np.zeros(output_size)

# Adam optimizer variables for Cross-Entropy Network
t_ce = 0
m_w1_ce = np.zeros_like(w1_ce)
v_w1_ce = np.zeros_like(w1_ce)
m_b1_ce = np.zeros_like(b1_ce)
v_b1_ce = np.zeros_like(b1_ce)
m_w2_ce = np.zeros_like(w2_ce)
v_w2_ce = np.zeros_like(w2_ce)
m_b2_ce = np.zeros_like(b2_ce)
v_b2_ce = np.zeros_like(b2_ce)

beta1 = 0.9
beta2 = 0.999
epsilon = 1e-8

# Activation functions
def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return (x > 0).astype(float)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Forward pass for Cross-Entropy Network
def forward_pass_ce(x):
    z1 = np.dot(x, w1_ce) + b1_ce
    a1 = relu(z1)
    z2 = np.dot(a1, w2_ce) + b2_ce
    a2 = sigmoid(z2)
    return z1, a1, z2, a2

# Backward pass for Cross-Entropy Network with Adam Optimizer
def backward_pass_ce(x, y, z1, a1, z2, a2, learning_rate):
    global w1_ce, b1_ce, w2_ce, b2_ce, t_ce, m_w1_ce, v_w1_ce, m_b1_ce,
v_b1_ce, m_w2_ce, v_w2_ce, m_b2_ce, v_b2_ce

    # Output layer error
    dz2 = a2 - y
    dw2 = np.dot(a1.T, dz2) / x.shape[0]
    db2 = np.sum(dz2, axis=0) / x.shape[0]

    # Hidden layer error
    da1 = np.dot(dz2, w2_ce.T)
    dz1 = da1 * relu_derivative(a1)
```

```
    dw1 = np.dot(x.T, dz1) / x.shape[0]
    db1 = np.sum(dz1, axis=0) / x.shape[0]

    # Adam optimizer update
    t_ce += 1
    # Update first moment estimates
    m_w1_ce = beta1 * m_w1_ce + (1 - beta1) * dw1
    m_b1_ce = beta1 * m_b1_ce + (1 - beta1) * db1
    m_w2_ce = beta1 * m_w2_ce + (1 - beta1) * dw2
    m_b2_ce = beta1 * m_b2_ce + (1 - beta1) * db2
    # Update second moment estimates
    v_w1_ce = beta2 * v_w1_ce + (1 - beta2) * (dw1 ** 2)
    v_b1_ce = beta2 * v_b1_ce + (1 - beta2) * (db1 ** 2)
    v_w2_ce = beta2 * v_w2_ce + (1 - beta2) * (dw2 ** 2)
    v_b2_ce = beta2 * v_b2_ce + (1 - beta2) * (db2 ** 2)
    # Compute bias-corrected moment estimates
    m_w1_ce_hat = m_w1_ce / (1 - beta1 ** t_ce)
    m_b1_ce_hat = m_b1_ce / (1 - beta1 ** t_ce)
    v_w1_ce_hat = v_w1_ce / (1 - beta2 ** t_ce)
    v_b1_ce_hat = v_b1_ce / (1 - beta2 ** t_ce)
    m_w2_ce_hat = m_w2_ce / (1 - beta1 ** t_ce)
    m_b2_ce_hat = m_b2_ce / (1 - beta1 ** t_ce)
    v_w2_ce_hat = v_w2_ce / (1 - beta2 ** t_ce)
    v_b2_ce_hat = v_b2_ce / (1 - beta2 ** t_ce)
    # Update parameters
    w1_ce -= learning_rate * m_w1_ce_hat / (np.sqrt(v_w1_ce_hat) +
epsilon)
    b1_ce -= learning_rate * m_b1_ce_hat / (np.sqrt(v_b1_ce_hat) +
epsilon)
    w2_ce -= learning_rate * m_w2_ce_hat / (np.sqrt(v_w2_ce_hat) +
epsilon)
    b2_ce -= learning_rate * m_b2_ce_hat / (np.sqrt(v_b2_ce_hat) +
epsilon)

# Training the Cross-Entropy Network
epochs = 50
learning_rate = 0.001
batch_size = 32

# Convert labels to binary classification (0 or 1)
y_train_ce = train_labels.reshape(-1, 1)
y_test_ce = test_labels.reshape(-1, 1)

# Lists to store cost histories
cost_history_ce = []
smooth_ce=[]
smooth_exp=[]
cost_history_exp = []
```

```
s_window=3 #επιλογη παραθύρου για smoothing

def accuracy_score(y_true, y_pred):
    return np.sum(y_true == y_pred) / len(y_true)

for epoch in range(epochs):
    # Cross-Entropy Network Training
    for i in range(0, train_images.shape[0], batch_size):
        x_batch = train_images[i:i+batch_size]
        y_batch = y_train_ce[i:i+batch_size]

        # Forward pass
        z1, a1, z2, a2 = forward_pass_ce(x_batch)

        # Compute cost
        cost = -np.mean(y_batch * np.log(a2 + epsilon) + (1 - y_batch)
* np.log(1 - a2 + epsilon))
        cost_history_ce.append(cost)

        # Backward pass
        backward_pass_ce(x_batch, y_batch, z1, a1, z2, a2,
learning_rate)
    if epoch% s_window==0:
        smooth_ce.append(np.mean(cost_history_ce[-s_window:]))
    # Compute training accuracy for Cross-Entropy Network
    _, _, _, train_output = forward_pass_ce(train_images)
    train_predictions = (train_output > 0.5).astype(int).flatten()
    train_accuracy = accuracy_score(train_labels, train_predictions)
    'print(f"Epoch {epoch + 1}/{epochs} - Cross-Entropy Training
Accuracy: {train_accuracy * 100:.2f}%")'

# Evaluate the Cross-Entropy Network on test data
_, _, _, test_output = forward_pass_ce(test_images)
test_predictions_ce = (test_output > 0.5).astype(int).flatten()
test_accuracy_ce = accuracy_score(test_labels, test_predictions_ce)
print(f"Cross-Entropy Network Error: {(1-test_accuracy_ce) *
100:.2f}%")
# Error percentage for Cross-Entropy Network
error_percentage_ce_0 = (np.sum((test_labels == 0) &
(test_predictions_ce != test_labels)) / np.sum(test_labels == 0)) * 100
error_percentage_ce_8 = (np.sum((test_labels == 1) &
(test_predictions_ce != test_labels)) / np.sum(test_labels == 1)) * 100
total_error_percentage_ce = ((np.sum(test_predictions_ce !=
test_labels)) / len(test_labels)) * 100
print("############## CROSS-ENTROPY NETWORK ##################")
print(f"error_percentage for 0:  {error_percentage_ce_0}%")
print(f"error_percentage for 8:  {error_percentage_ce_8}%")
print(f"total_error_percentage: {total_error_percentage_ce}%")
```

```
# Initialize neural network parameters for Exponential Loss Network
with Adam Optimizer
np.random.seed(0)
input_size = 784
hidden_size = 300
output_size = 1

# Weights initialization for Exponential Loss Network
w1_exp = np.random.normal(0, np.sqrt(1 / (input_size + hidden_size)),
(input_size, hidden_size))
b1_exp = np.zeros(hidden_size)
w2_exp = np.random.normal(0, np.sqrt(1 / (hidden_size + output_size)),
(hidden_size, output_size))
b2_exp = np.zeros(output_size)

# Adam optimizer variables
t_exp = 0
m_w1_exp = np.zeros_like(w1_exp)
v_w1_exp = np.zeros_like(w1_exp)
m_b1_exp = np.zeros_like(b1_exp)
v_b1_exp = np.zeros_like(b1_exp)
m_w2_exp = np.zeros_like(w2_exp)
v_w2_exp = np.zeros_like(w2_exp)
m_b2_exp = np.zeros_like(b2_exp)
v_b2_exp = np.zeros_like(b2_exp)

beta1 = 0.9
beta2 = 0.999
epsilon = 1e-8
# Backward pass for Exponential Loss Network with Adam Optimizer
def backward_pass_exp(x, y, z1, a1, z2, a2, learning_rate):
    global w1_exp, b1_exp, w2_exp, b2_exp, t_exp, m_w1_exp, v_w1_exp,
m_b1_exp, v_b1_exp, m_w2_exp, v_w2_exp, m_b2_exp, v_b2_exp

    # Output layer error
    dz2 = -0.5 * y * np.exp(-0.5 * y * a2)
    dw2 = np.dot(a1.T, dz2) / x.shape[0]
    db2 = np.sum(dz2, axis=0) / x.shape[0]
    # Hidden layer error
    da1 = np.dot(dz2, w2_exp.T)
    dz1 = da1 * relu_derivative(a1)
    dw1 = np.dot(x.T, dz1) / x.shape[0]
    db1 = np.sum(dz1, axis=0) / x.shape[0]
    # Adam optimizer update
    t_exp += 1
    # Update first moment estimates
    m_w1_exp = beta1 * m_w1_exp + (1 - beta1) * dw1
```

```
    m_b1_exp = beta1 * m_b1_exp + (1 - beta1) * db1
    m_w2_exp = beta1 * m_w2_exp + (1 - beta1) * dw2
    m_b2_exp = beta1 * m_b2_exp + (1 - beta1) * db2
    # Update second moment estimates
    v_w1_exp = beta2 * v_w1_exp + (1 - beta2) * (dw1 ** 2)
    v_b1_exp = beta2 * v_b1_exp + (1 - beta2) * (db1 ** 2)
    v_w2_exp = beta2 * v_w2_exp + (1 - beta2) * (dw2 ** 2)
    v_b2_exp = beta2 * v_b2_exp + (1 - beta2) * (db2 ** 2)
    # Compute bias-corrected moment estimates
    m_w1_exp_hat = m_w1_exp / (1 - beta1 ** t_exp)
    m_b1_exp_hat = m_b1_exp / (1 - beta1 ** t_exp)
    v_w1_exp_hat = v_w1_exp / (1 - beta2 ** t_exp)
    v_b1_exp_hat = v_b1_exp / (1 - beta2 ** t_exp)
    m_w2_exp_hat = m_w2_exp / (1 - beta1 ** t_exp)
    m_b2_exp_hat = m_b2_exp / (1 - beta1 ** t_exp)
    v_w2_exp_hat = v_w2_exp / (1 - beta2 ** t_exp)
    v_b2_exp_hat = v_b2_exp / (1 - beta2 ** t_exp)
    # Update parameters
    w1_exp -= learning_rate * m_w1_exp_hat / (np.sqrt(v_w1_exp_hat) +
epsilon)
    b1_exp -= learning_rate * m_b1_exp_hat / (np.sqrt(v_b1_exp_hat) +
epsilon)
    w2_exp -= learning_rate * m_w2_exp_hat / (np.sqrt(v_w2_exp_hat) +
epsilon)
    b2_exp -= learning_rate * m_b2_exp_hat / (np.sqrt(v_b2_exp_hat) +
epsilon)
# Training the Exponential Loss Network
epochs = 50
learning_rate = 0.001
batch_size = 32
# Training the Exponential Loss Network
# Convert labels to {-1, 1} for Exponential Loss Network
y_train_exp = np.where(train_labels == 1, 1, -1)
def forward_pass_exp(x):
    z1 = np.dot(x, w1_exp) + b1_exp
    a1 = relu(z1)
    z2 = np.dot(a1, w2_exp) + b2_exp
    a2 = z2  # No activation in output layer
    return z1, a1, z2, a2
for epoch in range(epochs):
    # Exponential Loss Network Training
    for i in range(0, train_images.shape[0], batch_size):
        x_batch = train_images[i:i+batch_size]
        y_batch = y_train_exp[i:i+batch_size].reshape(-1, 1)
        # Forward pass
        z1, a1, z2, a2 = forward_pass_exp(x_batch)
        # Compute cost
        cost = np.mean(np.exp(-0.5 * y_batch * a2))
```

```
        cost_history_exp.append(cost)
        # Backward pass
        backward_pass_exp(x_batch, y_batch, z1, a1, z2, a2,
learning_rate)
    if epoch% s_window==0:
        smooth_exp.append(np.mean(cost_history_exp[-s_window:]))
    # Compute training accuracy for Exponential Loss Network
    _, _, _, train_output = forward_pass_exp(train_images)
    train_predictions = (train_output > 0).astype(int).flatten()
    train_predictions = np.where(train_predictions == 0, -1, 1)
    train_accuracy = accuracy_score(y_train_exp, train_predictions)
    'print(f"Epoch {epoch + 1}/{epochs} - Exponential Loss Training
Accuracy: {train_accuracy * 100:.2f}%")'

# Evaluate the Exponential Loss Network on test data
_, _, _, test_output = forward_pass_exp(test_images)
test_predictions_exp = (test_output > 0).astype(int).flatten()
test_predictions_exp = np.where(test_predictions_exp == 0, -1, 1)
test_accuracy_exp = accuracy_score(np.where(test_labels == 1, 1, -1),
test_predictions_exp)
print(f"Exponential Network Error: {(1-test_accuracy_exp) * 100:.2f}%")
#display_misclassified_images(images, true_labels,
test_predictions_exp, "disapointment", num_images=10)
# Error percentage for Exponential Loss Network
error_percentage_exp_0 = (np.sum((test_labels == 0) &
(test_predictions_exp != np.where(test_labels == 0, -1, 1))) /
np.sum(test_labels == 0)) * 100
error_percentage_exp_8 = (np.sum((test_labels == 1) &
(test_predictions_exp != np.where(test_labels == 1, 1, -1))) /
np.sum(test_labels == 1)) * 100
total_error_percentage_exp = ((np.sum(test_predictions_exp !=
np.where(test_labels == 1, 1, -1))) / len(test_labels)) * 100
print("################## EXPONENTIAL ###############")
print(f"error_percentage for 0:  {error_percentage_exp_0}%")
print(f"error_percentage for 8:  {error_percentage_exp_8}%")
print(f"total_error_percentage:  {total_error_percentage_exp}%")
# Plot the cost histories for both networks

plt.plot(smooth_ce, label='Cross-Entropy Loss')
plt.plot(smooth_exp, label='Exponential Loss')
plt.xlabel(f'Iterations/{s_window}')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Comparison Between Cross-Entropy and Exponential Loss
Networks')
plt.show()
```