# Μηχανική Μάθηση - Εργασία 1

Σγουράκης Δημήτριος (ΑΜ: 1084584)

## Πρόβλημα 1

**Δεδομένα προβλήματος:**

- Τυχαίο διάνυσμα: $X = [x_1, x_2]$
- Υπόθεση $H_0$: $x_1$, $x_2$ ανεξάρτητες με συνάρτηση πυκνότητας πιθανότητας $f_0(x_1, x_2) = f_0(x_1) \times f_0(x_1)$, όπου: $f_0(x) \sim \mathcal{N}(0, 1)$
- Υπόθεση $H_1$: $x_1$, $x_2$ ανεξάρτητες με συνάρτηση πυκνότητας πιθανότητας $f_1(x_1, x_2) = f_1(x_1) \times f_1(x_1)$, όπου: $f_1(x) \sim 0.5\{\mathcal{N}(-1, 1) + \mathcal{N}(1, 1)\}$
- $P(H_0) = P(H_1)$

### Ερώτημα (a)

Στο συγκεκριμένο πρόβλημα οι αποφάσεις είναι δύο ($H_0$ και $H_1$). Οπότε τα πιθανά σενάρια είναι τα εξής 4:

- Σωστή απόφαση $H_0$ με κόστος απόφασης $C_{00}$
- Λανθασμένη απόφαση $H_0$ με κόστος απόφασης $C_{01}$
- Σωστή απόφαση $H_1$ με κόστος απόφασης $C_{11}$
- Λανθασμένη απόφαση $H_1$ με κόστος απόφασης $C_{10}$

Το βέλτιστο τεστ κατά Bayes απαιτεί τον υπολογισμό του λόγου πιθανοφάνειας:

$$L(X) = \frac{f_1(X)}{f_2(X)}$$

Συγκρίνοντας τον παραπάνω λόγο με ένα σταθερό κατώφλι, το οποίο συνιστάται από τα κόστη και της πιθανότητες της κάθε υπόθεσης να ισχύει, σχηματίζεται το βέλτιστο τεστ κατά Bayes, το οποίο ελαχιστοποιεί την πιθανότητα σφάλματος απόφασης:

$$Σύγκριση\ \frac{f_1(X)}{f_0(X)}\ με\ \frac{(C_{10} - C_{00})P(H_0)}{(C_{01} - C_{11})P(H_1)}$$

- Αν: $\frac{f_1(X)}{f_0(X)} > \frac{(C_{10}-C_{00})P(H_0)}{(C_{01}-C_{11})P(H_1)}$ ,τότε: $H_1$
- Αν: $\frac{f_1(X)}{f_0(X)} < \frac{(C_{10}-C_{00})P(H_0)}{(C_{01}-C_{11})P(H_1)}$ ,τότε: $H_0$
- Αν: $\frac{f_1(X)}{f_2(X)} = \frac{(C_{10}-C_{00})P(H_0)}{(C_{01}-C_{11})P(H_1)}$ ,τότε: $H_0$ ή $H_1$

Θεωρώντας ότι τα κόστη για σωστή απόφαση είναι μηδέν, τα κόστη για λανθασμένη απόφαση είναι ίσα με τη μονάδα και γνωρίζοντας ότι $P(H_0) = P(H_1)$, το βέλτιστο τεστ κατά Bayes γίνεται:

- Αν $\frac{f_1(X)}{f_0(X)} > 1 \Rightarrow H_1$
- Αν $\frac{f_1(X)}{f_0(X)} < 1 \Rightarrow H_0$
- Αν $\frac{f_1(X)}{f_0(X)} = 1 \Rightarrow H_0$ ή $H_1$

## Ερώτημα (b)

Χρησιμοποιώντας την Python και τη βιβλιοθήκη NumPy στο αρχείο **samples.py** δημιουργήθηκαν $10^6$ δείγματα για κάθε συνάρτηση πυκνότητας πιθανότητας ($f_0(x_0, x_1)$, $f_1(x_0, x_1)$). Γνωρίζοντας τις κατανομές υπολογίστηκαν οι δύο συναρτήσεις στο αρχείο **err_prob.py**:

- $f_0(x) \sim \mathcal{N}(0,1)$ , άρα:
$$f_0(x_0, x_1) = \frac{1}{\sqrt{2\pi}} e^{-\frac{{x_0}^2}{2}} \times \frac{1}{\sqrt{2\pi}} e^{-\frac{{x_1}^2}{2}}$$
- $f_1(x) \sim 0.5\{\mathcal{N}(-1,1) + \mathcal{N}(1,1)\}$ , άρα:
$$f_1(x_0, x_1) = 0.5 \times \frac{1}{\sqrt{2\pi}} \left( e^{-\frac{(x_0+1)^2}{2}} + e^{-\frac{(x_0-1)^2}{2}} \right) \times 0.5 \times \frac{1}{\sqrt{2\pi}} \left( e^{-\frac{(x_1+1)^2}{2}} + e^{-\frac{(x_1-1)^2}{2}} \right)$$

Τέλος, έγινε έλεγχος για κάθε δείγμα αν με βάση το βέλτιστο τεστ κατά Bayes η απόφαση που θα παρθεί τελικά είναι σωστή ή λανθασμένη. Μέσω αυτού του ελέγχου υπολογίστηκε το ποσοστό λανθασμένων αποφάσεων για κάθε σετ δεδομένων και τελικά η ολική πιθανότητα σφάλματος.

Το ποσοστό λανθασμένων αποφάσεων για το σετ δεδομένων με συνάρτηση πυκνότητάς πιθανότητας **$f_0(x_0, x_1)$** είναι **28.1825 %**. Το ποσοστό λανθασμένων αποφάσεων για το σετ δεδομένων με συνάρτηση πυκνότητάς πιθανότητας **$f_1(x_0, x_1)$** είναι **42.4282 %**. Άρα η ολική πιθανότητα σφάλματος είναι **35.3054 %**.

Παρακάτω φαίνονται και τα αποτελέσματα του κώδικα στο terminal:

```
The percentage of wrong decisions for the dataset with PDF f0 is:  28.1825 %
The percentage of wrong decisions for the dataset with PDF f1 is:  42.4282 %
The total probability of error is:  35.3054 %
```

*Εικόνα 1: Ερώτημα 1a - αποτελέσματα στο Python terminal*

## Ερώτημα (c)

Για αυτό το ερώτημα δημιουργήθηκε ένα νευρωνικό δίκτυο διαστάσεων 2 × 20 × 1, το οποίο παίρνει ως είσοδο το διάνυσμα $X(x_0, x_1)$ και δίνει ως έξοδο μια εκτίμηση του λόγου πιθανοφάνειας. Χρησιμοποιήθηκαν δύο μέθοδοι, cross-entropy και exponential, οι οποίες θα εξηγηθούν παρακάτω. Για να πραγματοποιηθεί η εκπαίδευση του νευρωνικού δικτύου ακολουθήθηκαν τα εξής τρία στάδια για κάθε ένα από τα 200 + 200 δείγματα (τα δείγματα είναι καταταγμένα σε τυχαία σειρά):

1. Forward propagation
2. Υπολογισμός κόστους
3. Backward propagation
4. Κανονικοποίηση παραγώγων
5. Υπολογισμός καινούργιων παραμέτρων

Το **forward propagation** είναι η διαδικασία υπολογισμού της εξόδου του νευρωνικού δικτύου δεδομένης της εισόδου του. Κάθε επίπεδο του δικτύου προκύπτει από μια συνάρτηση παραμέτρων με το προηγούμενο επίπεδο του δικτύου και μία μη-γραμμική συνάρτηση, η οποία εφαρμόζεται σε κάθε νευρώνα του επιπέδου. Οι παράμετροι του πρώτου επιπέδου είναι ο πίνακας βαρών $A1_{2\times20}$ και το διάνυσμα των offsets $B1_{1\times20}$. Οι παράμετροι του δευτέρου επιπέδου είναι ο πίνακας βαρών $A2_{20\times1}$ και το διάνυσμα των offsets $B2_{1\times1}$. Η μη-γραμμική συνάρτηση του πρώτου επιπέδου είναι η συνάρτηση ReLU(x), ενώ η μη-γραμμική συνάρτηση του δευτέρου επιπέδου εξαρτάται από τη μέθοδο, η οποία χρησιμοποιείται: σ(x) για cross-entropy και απλή γραμμική συνάρτηση f(x) = x για exponential. Οπότε τελικά ισχύουν οι παρακάτω σχέσεις:

$$Input\ layer: W_1 = A_1 \times X + B_1\ and\ Z_1 = ReLU(W_1)$$

$$Output\ layer: W_2 = A_2 \times Z_1 + B_2\ and\ Z_2 = g(W_2) = \begin{cases} \sigma(W_2)\ for\ cross-entropy \\ W_2\ for\ exponential \end{cases}$$

Η αρχικοποίηση των βαρών έγινε σύμφωνα με κανονική κατανομή:

$$\mathcal{N}\left(0, \frac{1}{n+m}\right), \qquad \text{όπου } n = 2 \text{ και } m = 20$$

Η αρχικοποίηση των offset είναι 0. Επιλέχτηκαν αυτές οι αρχικοποιήσεις, καθώς δεν είναι γνωστές εκ των προτέρων.

Ο **υπολογισμός κόστους** γίνεται, για να προσδιοριστεί εάν η μέθοδος συγκλίνει. Στην μέθοδο cross-entropy το κόστος εκτιμά την εκ των υστέρων πιθανότητα (posterior probability) και υπολογίζεται από τις συναρτήσεις:

$$\Phi(z) = \begin{cases} \varphi(z) = -\log(1-z)\ \gamma\iota\alpha\ H_0 \\ \psi(z) = -\log(z)\ \gamma\iota\alpha\ H_1 \end{cases}$$

Στην μέθοδο exponential το κόστος εκτιμά τον λόγο λογαριθμικής πιθανοφάνειας (log-likelihood ratio) και υπολογίζεται από τις συναρτήσεις:

$$\Phi(z) = \begin{cases} \varphi(z) = e^{0.5z} \; για \; H_0 \\ \psi(z) = e^{-0.5z} \; για \; H_1 \end{cases}$$

Κάθε 20 επαναλήψεις υπολογίζεται ο μέσος όρος του κόστους, ώστε να φαίνεται πιο ομαλή η σύγκλιση (αν υπάρχει).

Το **backward propagation** είναι οι διαδικασία υπολογισμού των μερικών παραγώγων των παραμέτρων του νευρωνικού δικτύου ($A_x$ και $B_x$) ξεκινώντας από την έξοδο και χρησιμοποιώντας τον κανόνα αλυσίδας. Είναι το πρώτο βήμα του αλγορίθμου Stochastic Gradient Descent (SGD):

$$\frac{\partial \Phi}{\partial Y} = \Phi'(Y)$$

$$\frac{\partial Y}{\partial W_2} = g'(W_2) = \begin{cases} \sigma(W_2) \cdot \big(1 - \sigma(W_2)\big) \; for \; cross - entropy \\ 1 \; for \; exponential \end{cases}$$

$$\frac{\partial W_2}{\partial Z_1} = A_2^T$$

$$\frac{\partial Z_1}{\partial W_1} = ReLU'(W_1) = \begin{cases} 0\,, & W_1 \leq 0 \\ 1, & W_1 > 0 \end{cases}$$

Έχοντας υπολογίσει τις παραπάνω τιμές και χρησιμοποιώντας τον κανόνα της αλυσίδας υπολογίζονται εύκολα και οι μερικοί παράγωγοι των παραμέτρων (Weights, Offsets):

$$\frac{\partial \Phi}{\partial A_2} = \frac{\partial \Phi}{\partial Y}\frac{\partial Y}{\partial W_2}\frac{\partial W_2}{\partial A_2} = \Phi'(Y) \times g'(W_2) \times Z_1^T$$

$$\frac{\partial \Phi}{\partial B_2} = \frac{\partial \Phi}{\partial Y}\frac{\partial Y}{\partial W_2}\frac{\partial W_2}{\partial B_2} = \Phi'(Y) \times g'(W_2) \times 1$$

$$\frac{\partial \Phi}{\partial A_1} = \frac{\partial \Phi}{\partial Y}\frac{\partial Y}{\partial W_2}\frac{\partial W_2}{\partial Z_1}\frac{\partial Z_1}{\partial W_1}\frac{\partial W_1}{\partial A_1} = \Phi'(Y) \times g'(W_2) \times ReLU'(W_1) \times X^T$$

$$\frac{\partial \Phi}{\partial B_1} = \frac{\partial \Phi}{\partial Y}\frac{\partial Y}{\partial W_2}\frac{\partial W_2}{\partial Z_1}\frac{\partial Z_1}{\partial W_1}\frac{\partial W_1}{\partial B_1} = \Phi'(Y) \times g'(W_2) \times ReLU'(W_1) \times 1$$

Η **κανονικοποίηση παραγώγων**, ενώ δεν είναι απαραίτητη, παράγει πιο σταθερές λύσεις. Για τον αλγόριθμο SGD χρησιμοποιήθηκε η κανονικοποίηση ADAM για κάθε μερική παράγωγο των παραμέτρων:

$$P_{[A_kB_k]}(t) = (1 - \lambda)P_{[A_kB_k]}(t-1) + \lambda \left(\nabla_{[A_kB_k]_{t-1}} \Phi(Y_t)\right)^{.2}, \text{όπου } \lambda \ll 1$$

$$P_{[A_kB_k]}(1) = \left(\nabla_{[A_kB_k]_0}\Phi(Y_1)\right)^{.2}, \qquad \alpha\rho\chi\iota\kappa o\pi o\acute{\iota}\eta\sigma\eta \ \tau\eta\varsigma \ \epsilon\kappa\tau\acute{\iota}\mu\eta\sigma\eta\varsigma \ \iota\sigma\chi\acute{\upsilon}o\varsigma$$

Το τελευταίο βήμα της εκπαίδευσης και του αλγορίθμου SGD είναι ο **υπολογισμός των καινούργιων παραμέτρων** μέσω του παρακάτω τύπου:

$$[A_kB_k]_t = [A_kB_k]_{t-1} - \mu\nabla_{[A_kB_k]_{t-1}}\Phi(Y_t) \Big/ . \sqrt{c + P_{[A_kB_k]}(t)} \quad \text{όπου } \mu = \rho\upsilon\theta\mu\acute{o}\varsigma \ \mu\alpha\theta\acute{\eta}\sigma\epsilon\omega\varsigma$$

Το c είναι μια μικρή σταθερά, ώστε να μην γίνεται διαίρεση με το μηδέν.

Κάθε φορά που εξαντλούνται τα δείγματα, ανακυκλώνονται και μπαίνουν σε τυχαία σειρά (νέα εποχή). Αυτός ο κύκλος επαναλαμβάνεται αρκετές φορές, ώστε να υπάρχει σύγκλιση. Έτσι ολοκληρώθηκε η διαδικασία εκπαίδευσης του νευρωνικού δικτύου.

Για την εξέταση του ποσοστού σφάλματος των δύο δικτύων χρησιμοποιήθηκαν τα $10^6$+$10^6$ δείγματα του ερωτήματος (b). Χρησιμοποιώντας τις δύο μεθόδους εκπαίδευσης στο αρχείο **gen_nn_1c.py** και εξετάζοντας κάθε ένα από τα δύο νευρωνικά δίκτυα με τα ίδια δείγματα υπολογίστηκαν τα παρακάτω σφάλματα:

```
H0 error:  27.0624 %
H1 error:  44.4572 %
Total error using the cross-entropy method:  35.7598 %
```

*Εικόνα 2: Πρόβλημα 1c - cross-entropy error*

```
H0 error:  29.7605 %
H1 error:  41.8829 %
Total error using the exponential method:  35.8217 %
```

*Εικόνα 3: Πρόβλημα 1c - exponential error*

Από τα αποτελέσματα φαίνεται ότι και οι δύο μέθοδοι έχουν παραπλήσιο ποσοστό λανθασμένων αποφάσεων, όταν εφαρμόζονται πάνω στα ίδια δεδομένα. Για τα ίδια δεδομένα το βέλτιστο τεστ κατά Bayes είχε πιθανότητα σφάλματος 35.3054%. Τα δύο νευρωνικά δίκτυα πλησίασαν το ποσοστό αυτό αλλά δεν έπεσαν κάτω από αυτό, καθώς αυτή η πιθανότητα σφάλματος είναι η βέλτιστη που μπορεί να επιτευχθεί.

Το νευρωνικό δίκτυο υλοποιείται ως κλάση στο αρχείο **neural_net.py**

# Πρόβλημα 2

## Δεδομένα προβλήματος

- Βάση δεδομένων MNIST – αρχεία αριθμών 0 και 8
- Εικόνες 28×28 pixel

## Επίλυση προβλήματος

Για αυτό το πρόβλημα απομονώθηκαν τα ψηφία 0 και 8 της βάσης MNIST. Δημιουργήθηκαν νευρωνικά δίκτυα 728×30×1, τα οποία παίρνουν ως είσοδο τα 28×28 pixels της κάθε εικόνας που υπάρχει στα datasets και επιστρέφουν ως έξοδο τον λόγο πιθανοφάνειας για τα ψηφία 0 και 8. Πιο συγκεκριμένα δημιουργήθηκαν δύο νευρωνικά δίκτυα: Ένα δίκτυο, στο οποίο εφαρμόστηκε η μέθοδος cross-entropy και ένα δίκτυο, στο οποίο εφαρμόστηκε η μέθοδος exponential. Ο αλγόριθμος του προηγούμενου προβλήματος ακολουθήθηκε και σε αυτό το πρόβλημα. Τα παρακάτω αποτελέσματα βγήκαν χρησιμοποιώντας 11774 δείγματα για την εκπαίδευση του νευρωνικού δικτύου και 980 + 974 δείγματα για την εξέταση των ποσοστών σφάλματος του κάθε δικτύου στο αρχείο **gen_nn_2.py**:

| Μέθοδος | Cross-entropy | Exponential |
|---|---|---|
| Σφάλμα στο ψηφίο 0 | 0.3061 % | 0.6122 % |
| Σφάλμα στο ψηφίο 8 | 0.0000 % | 1.4374 % |
| Ολικό σφάλμα | 0.1535 % | 1.0235 % |

Αποτελέσματα στο terminal:

```
Error rate for numeral 0:  0.3061 %
Error rate for numeral 8:  0.0 %
Total error using the cross-entropy method:  0.1535 %
```

*Εικόνα 4: Πρόβλημα 2 - cross-entropy*

```
Error rate for numeral 0:  0.6122 %
Error rate for numeral 8:  1.4374 %
Total error using the exponential method:  1.0235 %
```

*Εικόνα 5: Πρόβλημα 2 – exponential*

Το νευρωνικό δίκτυο υλοποιείται ως κλάση στο αρχείο **neural_net.py**

## Αρχεία υλοποίησης Python

**sample.py**

```python
import numpy as np
import json

size_b, size_c = 1000000, 200

# Sample for f1(x)
def sample_f1(size):
    choice = np.random.normal(0, 1, size)
    samples1 = np.random.normal(-1, 1, size)
    samples2 = np.random.normal(1, 1, size)
    samples = np.where(choice < 0, samples1, samples2)
    return samples

if __name__ == "__main__":
    # Create 10e6 samples to compute total probability error
    f0_x0_samples_b = np.random.normal(0, 1, size_b)
    f0_x1_samples_b = np.random.normal(0, 1, size_b)
    f1_x0_samples_b = sample_f1(size_b)
    f1_x1_samples_b = sample_f1(size_b)

    # Create 200 samples to train neural networks
    f0_x0_samples_c = np.random.normal(0, 1, size_c)
    f0_x1_samples_c = np.random.normal(0, 1, size_c)
    f1_x0_samples_c = sample_f1(size_c)
    f1_x1_samples_c = sample_f1(size_c)

    # Clear json files
    open('f0_x0_samples_b.json', 'w').close()
    open('f0_x1_samples_b.json', 'w').close()
    open('f1_x0_samples_b.json', 'w').close()
    open('f1_x1_samples_b.json', 'w').close()
    open('f0_x0_samples_c.json', 'w').close()
    open('f0_x1_samples_c.json', 'w').close()
    open('f1_x0_samples_c.json', 'w').close()
    open('f1_x1_samples_c.json', 'w').close()

    # Fill json files with samples
    with open('f0_x0_samples_b.json', 'a') as f: json.dump(f0_x0_samples_b.tolist(), f)
    with open('f0_x1_samples_b.json', 'a') as f: json.dump(f0_x1_samples_b.tolist(), f)
    with open('f1_x0_samples_b.json', 'a') as f: json.dump(f1_x0_samples_b.tolist(), f)
    with open('f1_x1_samples_b.json', 'a') as f: json.dump(f1_x1_samples_b.tolist(), f)
```

```
    with open('f0_x0_samples_c.json', 'a') as f: json.dump(f0_x0_samples_c.tolist(), f)
    with open('f0_x1_samples_c.json', 'a') as f: json.dump(f0_x1_samples_c.tolist(), f)
    with open('f1_x0_samples_c.json', 'a') as f: json.dump(f1_x0_samples_c.tolist(), f)
    with open('f1_x1_samples_c.json', 'a') as f: json.dump(f1_x1_samples_c.tolist(), f)
```

**err_prob.py**

```
import numpy as np
import json

from sample import size_b

factor = 1 / np.sqrt(2*np.pi)

# Calculate f0(x)
def f0(x):
    return factor * np.exp(-0.5 * x**2)

# Calculate f1(x)
def f1(x):
    return 0.5 * factor * (np.exp(-0.5 * (x + 1)**2) + np.exp(-0.5 * (x - 1)**2))

# Calculate f0(x1, x2)
def f_0(x1, x2):
    return f0(x1) * f0(x2)

# Calulate f1(x1, x2)
def f_1(x1, x2):
    return f1(x1) * f1(x2)

# Calculate error for dataset from f0
def h0_error(samples1, samples2, size):
    count = 0
    for i in range(size):
        if f_0(samples1[i], samples2[i]) < f_1(samples1[i], samples2[i]): count = count
+ 1
    return count / size

# Calculate error for dataset from f1
def h1_error(samples1, samples2, size):
    count = 0
    for i in range(size):
        if f_0(samples1[i], samples2[i]) > f_1(samples1[i], samples2[i]): count = count
+ 1
```

```
    return count / size

with open('f0_x0_samples_b.json', 'r') as f: f0_x0_samples = json.load(f)
with open('f0_x1_samples_b.json', 'r') as f: f0_x1_samples = json.load(f)
with open('f1_x0_samples_b.json', 'r') as f: f1_x0_samples = json.load(f)
with open('f1_x1_samples_b.json', 'r') as f: f1_x1_samples = json.load(f)

err_0 = h0_error(f0_x0_samples, f0_x1_samples, size_b)
err_1 = h1_error(f1_x0_samples, f1_x1_samples, size_b)
total_err = (err_0 + err_1) / 2

print("The percentage of wrong decisions for the dataset with PDF f0 is: ",
np.round(err_0 * 100, 4), "%")
print("The percentage of wrong decisions for the dataset with PDF f1 is: ",
np.round(err_1 * 100, 4), "%")
print("The total probability of error is: ", np.round(total_err * 100, 4), "%")
```

**neural_net.py**

```
import numpy as np
from sklearn import datasets

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

def ReLU(X):
    return np.maximum(0, X)

class NeuralNetwork:
    def __init__(self, in_size: int, hid_size: int, method: str):
        self.in_size = in_size      #n
        self.hid_size = hid_size    #m
        self.out_size = 1           #k
        self.method = method
        self.cost = np.array([])
        self.converge = np.array([])
        self.error = np.array([])
        self.it = 0

        # Weights (A, B --> matrix) and offsets (a, b --> vector) of each layer
        self.A1 = np.matrix(np.random.normal(0, 1/(self.in_size + self.hid_size),
(self.in_size, self.hid_size))) # n x m
        self.A2 = np.matrix(np.random.normal(0, 1/(self.hid_size + self.out_size),
(self.hid_size, self.out_size))) # m x k
```

```
        self.B1 = np.matrix(np.zeros(self.hid_size)) # 1 x m
        self.B2 = np.matrix(np.array([0])) # 1 x k

        # Weight gradients
        self.dA1 = 0
        self.dA2 = 0
        self.dB1 = 0
        self.dB2 = 0

        # Weight gradient powers
        self.P_dA1 = 0
        self.P_dA2 = 0
        self.P_dB1 = 0
        self.P_dB2 = 0
        self.lamda = 0.000001
        self.c = 0.00000001

    # Propagate forward through the neural network
    def forward(self, X: np.ndarray) -> np.matrix:
        """Propagate forward through the neural network.\n
        Uses a ReLU activation function for the hidden layer and a sigmoid activation
        function for the output layer if self.method == "CEM".\n
        Returns the output layer of the network."""

        self.Z0 = np.matrix(X)
        #W1 = A1*Z0 + B1 --> Hidden layer output before non-linear function 1 x m
        self.W1 = np.dot(self.Z0, self.A1) + self.B1
        #Z1 = ReLU(W1) --> Hidden layer output after non-linear function 1 x m
        self.Z1 = ReLU(self.W1)
        #W2 = A2*Z1 + B2 --> Output layer output before non-linear function 1 x k
        self.W2 = np.dot(self.Z1, self.A2) + self.B2
        #Z2 = σ(W2) or W2 --> Output layer output after non-linear function 1 x k
        if self.method == "CEM":
            self.Z2 = sigmoid(self.W2)
        elif self.method == "EXP":
            self.Z2 = self.W2
        self.it += 1
        return self.Z2

    # Calculate the cost value
    def calc_cost(self, y: float):
        """Calculates the cost value of the output based on the correct value
        and the method.\n
        For the CEM method ϕ(z) =  -log(1 - z) and ψ(z) =  -log(z)\n
        For the EXP method ϕ(z) = e^(0.5z) and ψ(z) = e^(-0.5z)"""
```

```
    if self.method == "CEM":
        if y == 0:
            # ϕ(z) =  -log(1 - z)
            c = - np.log10(1 - self.Z2[0, 0])
        else:
            # ψ(z) =  -log(z)
            c = - np.log10(self.Z2[0, 0])
    elif self.method == "EXP":
        if y == 0:
            # ϕ(z) = e^(0.5z)
            c = np.exp(0.5* self.Z2[0, 0])
        else:
            # ψ(z) = e^(-0.5z)
            c = np.exp(-0.5* self.Z2[0, 0])
    self.cost = np.append(self.cost, c)

# Compute the gradients
def backward(self, y: float):
    """Propagate backwards through the neueral network calculating the gradients
    of the weights and the offsets with respect to the cost function (L(z)).\n
    self.dW1 --> dL / dW2, self.dW2 --> dL / dW1, self.db1 --> dL / db1,
    self.db1 --> dL / db2"""

    if self.method == "CEM":
        if y == 0:
            dZ2 = 1 / (np.log(10) * (1 - self.Z2[0, 0]))
        else:
            dZ2 = - 1 / (np.log(10) * self.Z2[0, 0])
        dW2 = np.matrix(dZ2 * sigmoid(self.W2) * (1 - sigmoid(self.W2)))
    elif self.method == "EXP":
        if y == 0:
            dZ2 = 0.5 * np.exp(0.5* self.Z2[0, 0])
        else:
            dZ2 = -0.5 * np.exp(-0.5 * self.Z2[0, 0])
        dW2 = np.matrix(dZ2)
    self.dA2 = np.dot(self.Z1.T, dW2)
    self.dB2 = dW2
    dZ1 = np.dot(dW2, self.A2.T)
    dW1 = np.multiply(dZ1, np.where(self.W1 > 0, 1, 0))
    self.dA1 = np.dot(self.Z0.T, dW1)
    self.dB1 = dW1

# Compute the power estimate of each gradient for the first iteration
def power_init(self):
```

```
        """Compute the power estimate of each gradient for the first iteration."""

        self.P_dA1 = np.square(self.dA1)
        self.P_dA2 = np.square(self.dA2)
        self.P_dB1 = np.square(self.dB1)
        self.P_dB2 = np.square(self.dB2)

    # Compute the power estimate of each gradient
    def power_get(self):
        """Compute the power estimate of each gradient after the first iteration."""

        self.P_dA1 = (1 - self.lamda) * self.P_dA1 + self.lamda * np.square(self.dA1)
        self.P_dA2 = (1 - self.lamda) * self.P_dA2 + self.lamda * np.square(self.dA2)
        self.P_dB1 = (1 - self.lamda) * self.P_dB1 + self.lamda * np.square(self.dB1)
        self.P_dB2 = (1 - self.lamda) * self.P_dB2 + self.lamda * np.square(self.dB2)

    # Update the weights and the offsets
    def update(self, m: float):
        """Update the weights and the offsets using the calculated gradients
        and power estimations.\n
        m = learning rate"""

        self.A1 = self.A1 - np.divide(np.multiply(m, self.dA1), np.add(self.c,
np.sqrt(self.P_dA1)))
        self.A2 = self.A2 - np.divide(np.multiply(m, self.dA2), np.add(self.c,
np.sqrt(self.P_dA2)))
        self.B1 = self.B1 - np.divide(np.multiply(m, self.dB1), np.add(self.c,
np.sqrt(self.P_dB1)))
        self.B2 = self.B2 - np.divide(np.multiply(m, self.dB2), np.add(self.c,
np.sqrt(self.P_dB2)))

    # Train the neural network for problem 1
    def train1(self, samples0: np.ndarray, samples1: np.ndarray, outputs: np.ndarray,
m: float, epochs=100):
        """Train the neural network for problem 1."""

        for ep in range(epochs):
            r = np.arange(len(outputs))
            np.random.shuffle(r)
            samples0 = samples0[r]
            samples1 = samples1[r]
            outputs = outputs[r]
            cost20 = 0
            counter = 0
```

```
            for sample in range(len(outputs)):
                X = np.array([samples0[sample], samples1[sample]])
                self.forward(X)
                self.calc_cost(outputs[sample])
                self.backward(outputs[sample])
                if ep == 0 and sample == 0:
                    self.power_init()
                else:
                    self.power_get()
                self.update(m)
                if counter == 19:
                    cost20 = np.mean(self.cost)
                    self.cost = np.array([])
                    counter = 0
                    self.converge = np.append(self.converge, cost20)
                counter += 1

    # Train the neural network for problem 2
    def train2(self, samples, vals, m, epochs):
        """Train the neural network for problem 2 --> MNIST dataset\n
        This time X is 784"""

        for ep in range(epochs):
            r = np.arange(len(samples))
            np.random.shuffle(r)
            samples_rand = samples.iloc[r]
            vals_rand = vals.iloc[r]
            cost20 = 0
            counter = 0

            for sample in range(len(samples_rand)):
                x = np.divide(np.array([samples_rand.iloc[sample]])[0], 255)
                if vals_rand.iloc[sample] == 8:
                    y = 1
                else:
                    y = 0
                self.forward(x)
                self.calc_cost(y)
                self.backward(y)
                if ep == 0 and sample == 0:
                    self.power_init()
                else:
                    self.power_get()
                self.update(m)
                if counter == 19:
```

```
                    cost20 = np.mean(self.cost)
                    self.cost = np.array([])
                    counter = 0
                    self.converge = np.append(self.converge, cost20)
                counter += 1

    # Apply the neural network to generated samples for problem 1
    def test1(self, samplesH0: np.ndarray, samplesH1: np.ndarray):
        """Apply the neural network to generated samples for problem 1 and print
        the error percentages.\n
        samplesH0: 2xn array with n samples from hypothesis H0\n
        samplesH1: 2xn array with n samples from hypothesis H1"""

        errH0 = 0
        errH1 = 0

        if self.method == "CEM":
            for sample in range(len(samplesH0[0])):
                z = self.forward(np.array([samplesH0[0][sample],
samplesH0[1][sample]]))
                if z > 0.5: errH0 += 1

            for sample in range(len(samplesH1[0])):
                z = self.forward(np.array([samplesH1[0][sample],
samplesH1[1][sample]]))
                if z < 0.5: errH1 += 1

        elif self.method == "EXP":
            for sample in range(len(samplesH0[0])):
                z = self.forward(np.array([samplesH0[0][sample],
samplesH0[1][sample]]))
                if z > 0: errH0 += 1

            for sample in range(len(samplesH1[0])):
                z = self.forward(np.array([samplesH1[0][sample],
samplesH1[1][sample]]))
                if z < 0: errH1 += 1


        errorH0 = errH0 / len(samplesH0[0])
        errorH1 = errH1 / len(samplesH1[0])
        total_error = (errorH0 + errorH1) / 2

        print("H0 error: ", np.round(errorH0 * 100, 4), "%")
        print("H1 error: ", np.round(errorH1 * 100, 4), "%")
```

```
        if self.method == "CEM":
            print("Total error using the cross-entropy method: ", np.round(total_error
* 100, 4), "%")
        elif self.method == "EXP":
            print("Total error using the exponential method: ", np.round(total_error *
100, 4), "%")

    def test2(self, samples0, samples8, vals0, vals8):
        """Apply the neural network to generated samples for problem 1 and print
        the error percentages.\n
        samples: mnist database testing samples\n
        vals: mnist database corresponding values"""

        err0 = 0
        err8 = 0

        if self.method == "CEM":
            for sample in range(len(samples0)):
                x = np.divide(np.array([samples0.iloc[sample]])[0], 255)
                z = self.forward(x)
                if z > 0.5 and vals0.iloc[sample] == 0: err0 += 1

            for sample in range(len(samples8)):
                x = np.divide(np.array([samples8.iloc[sample]])[0], 255)
                z = self.forward(x)
                if z < 0.5 and vals8.iloc[sample] == 0: err8 += 1

        elif self.method == "EXP":
            for sample in range(len(samples0)):
                x = np.divide(np.array([samples0.iloc[sample]])[0], 255)
                z = self.forward(x)
                if z > 0 and vals0.iloc[sample] == 0: err0 += 1

            for sample in range(len(samples8)):
                x = np.divide(np.array([samples8.iloc[sample]])[0], 255)
                z = self.forward(x)
                if z < 0 and vals8.iloc[sample] == 8: err8 += 1

        error0 = err0 / len(samples0)
        error8 = err8 / len(samples8)
        total_error = (error0 * len(samples0) + error8 * len(samples8)) /
(len(samples0) + len(samples8))

        print("Error rate for numeral 0: ", np.round(error0 * 100, 4), "%")
        print("Error rate for numeral 8: ", np.round(error8 * 100, 4), "%")
```

```
        if self.method == "CEM":
            print("Total error using the cross-entropy method: ", np.round(total_error
* 100, 4), "%")
        elif self.method == "EXP":
            print("Total error using the exponential method: ", np.round(total_error *
100, 4), "%")
```

**gen_nn_1c.py**

```
from neural_net import NeuralNetwork
import numpy as np
import json
import time

# Get training samples
with open("f0_x0_samples_c.json", "r") as f: samples0x0 = np.array(json.load(f))
with open("f0_x1_samples_c.json", "r") as f: samples0x1 = np.array(json.load(f))
with open("f1_x0_samples_c.json", "r") as f: samples1x0 = np.array(json.load(f))
with open("f1_x1_samples_c.json", "r") as f: samples1x1 = np.array(json.load(f))

# Get testing samples
with open("f0_x0_samples_b.json", "r") as f: samples0x0test = np.array(json.load(f))
with open("f0_x1_samples_b.json", "r") as f: samples0x1test = np.array(json.load(f))
with open("f1_x0_samples_b.json", "r") as f: samples1x0test = np.array(json.load(f))
with open("f1_x1_samples_b.json", "r") as f: samples1x1test = np.array(json.load(f))

# Process training samples
samples_x0 = np.concatenate([samples0x0, samples1x0])
samples_x1 = np.concatenate([samples0x1, samples1x1])
outputs = np.concatenate([np.zeros(len(samples0x0)), np.ones(len(samples1x0))])

# Process testing samples
samples_H0_test = np.vstack([samples0x0test, samples0x1test])
samples_H1_test = np.vstack([samples1x0test, samples1x1test])

# Create neural network (2x20x1)
nn_cem = NeuralNetwork(2, 20, "CEM")
nn_exp = NeuralNetwork(2, 20, "EXP")
ep = 400

# Best m:
# CEM --> 0.0001, EXP --> 0.0001

t0 = time.time()
```

```
nn_cem.train1(samples_x0, samples_x1, outputs, m=0.0001, epochs=ep)
nn_cem.test1(samples_H0_test, samples_H1_test)
t1 = time.time()
print("Time (CEM): ", t1 - t0, " seconds")

t0 = time.time()
nn_exp.train1(samples_x0, samples_x1, outputs, m=0.0001, epochs=ep)
nn_exp.test1(samples_H0_test, samples_H1_test)
t1 = time.time()
print("Time (EXP): ", t1 - t0, " seconds")
```

**gen_nn_2.py**

```
from neural_net import NeuralNetwork
from sklearn import datasets
import time


mnist = datasets.fetch_openml('mnist_784')
x_train = mnist.data[:60000]
x_test = mnist.data[60000:]
y_train = mnist.target.astype(int)[:60000]
y_test = mnist.target.astype(int)[60000:]

mask_train = (y_train == 0) | (y_train == 8)
train = x_train[mask_train]
vals_train = y_train[mask_train]

mask_test0 = (y_test == 0)
mask_test8 = (y_test == 8)
test0 = x_test[mask_test0]
test8 = x_test[mask_test8]
vals_test0 = y_test[mask_test0]
vals_test8 = y_test[mask_test8]

print("MNIST loaded")

# Create neural network (784x20x1)
nn_cem = NeuralNetwork(784, 300, "CEM")
nn_exp = NeuralNetwork(784, 300, "EXP")
ep = 1

# Best m:
# CEM --> 0.0001, EXP --> 0.0001
```

```
t0 = time.time()
print("Traing using CEM method...")
nn_cem.train2(train, vals_train, m=0.0001, epochs=ep)
print("Testing the neural network...")
nn_cem.test2(test0, test8, vals_test0, vals_test8)
t1 = time.time()
print("Time (CEM): ", t1 - t0, " seconds")

t0 = time.time()
print("Traing using EXP method...")
nn_exp.train2(train, vals_train, m=0.0001, epochs=ep)
print("Testing the neural network...")
nn_exp.test2(test0, test8, vals_test0, vals_test8)
t1 = time.time()
print("Time (EXP): ", t1 - t0, " seconds")
```