Μηχανική Μάθηση - Εργασία 3

Σγουράκης Δημήτριος (ΑΜ: 1084584)

Πρόβλημα 3.1

Σε αυτό το πρόβλημα δίνονται οι τυχαίες μεταβλητές $\mathcal{X}, \mathcal{Y}$, οι οποίες συνδέονται μέσω της σχέσεως:

$$Y = 0.8X + W \ , \qquad W = N(0,1)$$

Σκοπός του προβλήματος είναι ο υπολογισμός των παρακάτω δεσμευμένων μέσων όρων:

$$E_{\mathcal{Y}}{}^{1}[G_1(\mathcal{Y}) = \mathcal{Y}|\mathcal{X} = X] \quad \kappa\alpha\iota \quad E_{\mathcal{Y}}{}^{2}[G_2(\mathcal{Y}) = \min\{1, max\{-1, \mathcal{Y}\}\}|\mathcal{X} = X]$$

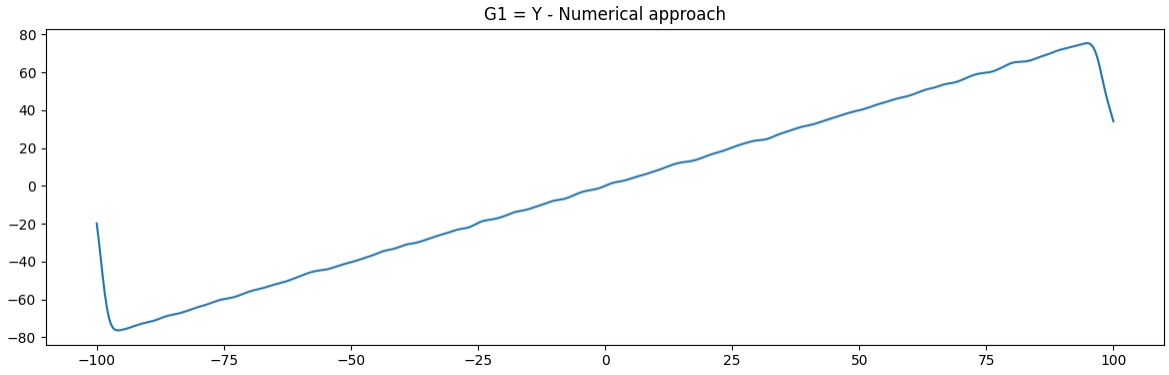
Αριθμητική προσέγγιση

Δημιουργήθηκαν 500 δείγματα του X και του W και υπολογίστηκαν τα αντίστοιχα δείγματα του Y. Έτσι δημιουργήθηκαν 500 ζευγάρια (X, Y), τα οποία χρησιμοποιήθηκαν για τον υπολογισμό των δύο διανυσμάτων G μήκους 500 και των δύο πινάκων $\mathcal{F}$ διαστάσεων 500×500. Δομή σειράς i πίνακα $\mathcal{F}$:

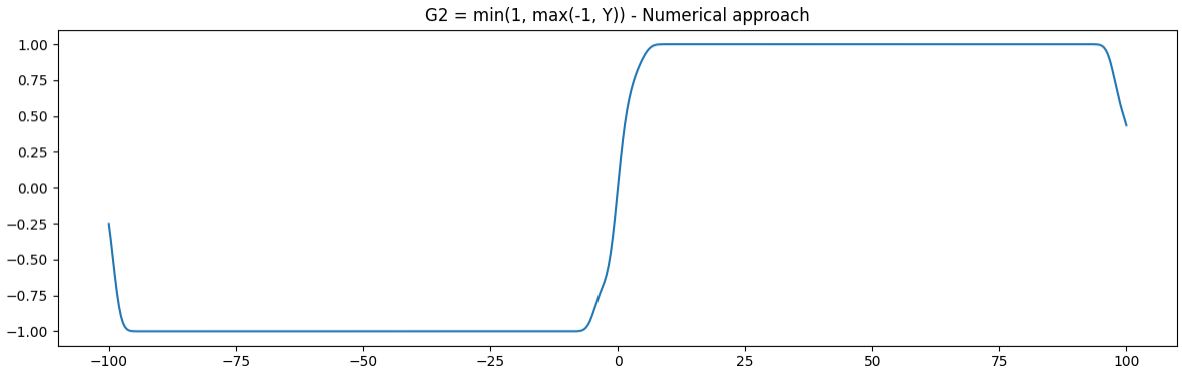
$$\left[\frac{H(Y_1|X_i) - H(Y_0|X_i)}{2} \quad \frac{H(Y_2|X_i) - H(Y_0|X_i)}{2} \quad \dots \quad \frac{H(Y_{500}|X_i) - H(Y_{498}|X_i)}{2} \quad \frac{H(Y_{500}|X_i) - H(Y_{499}|X_i)}{2}\right]$$

Τελικά τα διανύσματα V_1 και V_2 υπολογίζονται μέσω του γινομένου: $V_i = \mathcal{F}_i \cdot G_i$

Η δειγματοληψία και ο υπολογισμός αποτελεσμάτων υλοποιήθηκαν στο αρχείο **cond_exp_numerical.py**. Η παρουσίαση των αποτελεσμάτων και ο υπολογισμός της κλίσης της ευθείας $V_1(X)$ έγιναν στο αρχείο **results_1.py**.



Εικόνα 1: G1(Y)



Εικόνα 2: G2(Y)

Προσέγγιση με νευρωνικά δίκτυα

Καινούργια 500 δείγματα X (N(0, 20): επιλέχτηκε διασπορά 20, ώστε να φαίνεται καλύτερα η γραφική παράσταση των δειγμάτων) και G χρησιμοποιήθηκαν σε δύο νευρωνικά δίκτυα (Ένα για κάθε δεσμευμένο μέσο όρο) διαστάσεων 1×50×1, τα οποία εκπαιδεύτηκαν με αλγόριθμο Gradient Descent (GD).

Πιο συγκεκριμένα για κάθε επανάληψη του αλγορίθμου υπολογίζεται για κάθε δείγμα X η έξοδος του νευρωνικού δικτύου Z (forward pass), το κόστος J(u) και τα Gradients των παραμέτρων του νευρωνικού δικτύου. Έπειτα υπολογίζεται η μέση τιμή του κόστους από όλα τα δείγματα X, η οποία και πρέπει να ελαχιστοποιηθεί, και εφαρμόζεται ο τύπος του αλγορίθμου GD για των υπολογισμό των νέων παραμέτρων. Ο αλγόριθμος:

$$J(u) = E_{X,Y}\left[\varphi\left(u(X)\right) + G(Y)\psi\left(u(X)\right)\right]$$

$$\theta_t = \theta_{t-1} - \mu \sum_{i=1}^{n} \left[G(Y_i) - \omega\left(u(X_i, \theta_{t-1})\right)\right] \rho\left(u(X_i, \theta_{t-1})\right) \nabla_\theta \left(u(X_i, \theta_{t-1})\right)$$

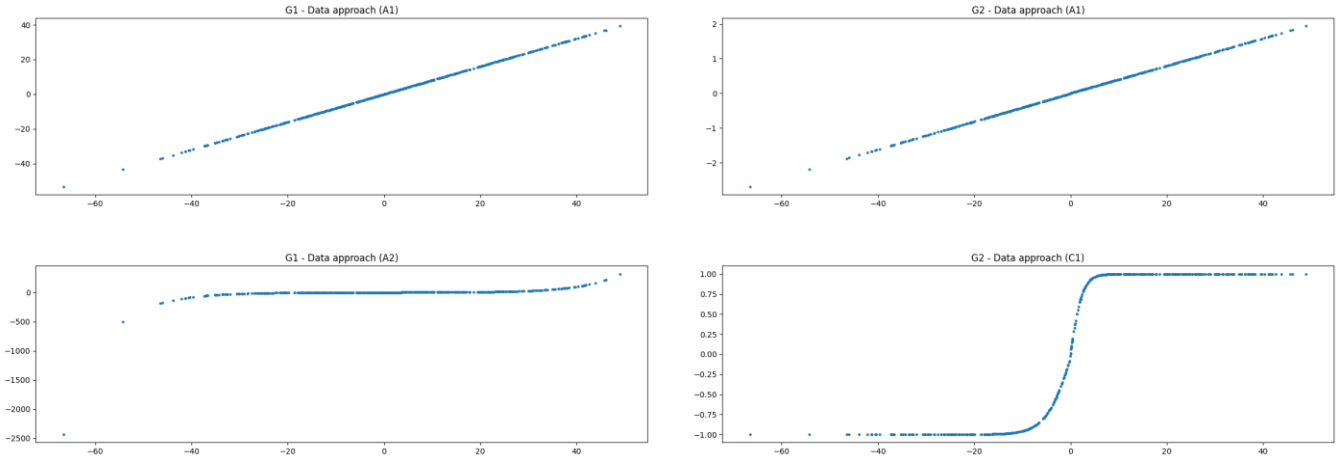
Οι συναρτήσεις φ, ψ, ρ, ω διαφέρουν ανάλογα με τη μέθοδο που χρησιμοποιείται. Σε αυτό το ερώτημα χρησιμοποιήθηκαν οι μέθοδοι A1, A2 για την G1 και A1, C1 με όρια [-1, 1] για την G2. Ο παραπάνω αλγόριθμος επαναλήφθηκε 100 φορές για G1(A1, A2) και G2(A1) και 200 φορές για G2(C1). Το μ (ρυθμός μαθήσεως) που χρησιμοποιήθηκε για G1 είναι 0.001, για G2(A1) είναι 0.01 και για G2(C1) είναι 0.05. Τα δύο τελικά νευρωνικά δίκτυα χρησιμοποιήθηκαν για τον υπολογισμό των τελικών διανυσμάτων V μέσω της συνάρτησης ω. Η δειγματοληψία, η δημιουργία των νευρωνικών δικτύων και η εκπαίδευσή τους υλοποιήθηκαν στον αρχείο **cond_exp_data.py**. Η παρουσίαση των αποτελεσμάτων και ο υπολογισμός της κλίσης της ευθείας $V_1(X)$ έγιναν στο αρχείο **results_1.py**. Η υλοποίηση του νευρωνικού δικτύου βρίσκεται στο αρχείο **neural_net.py**.

Τα αποτελέσματα των κλίσεων για την συνάρτηση G1:

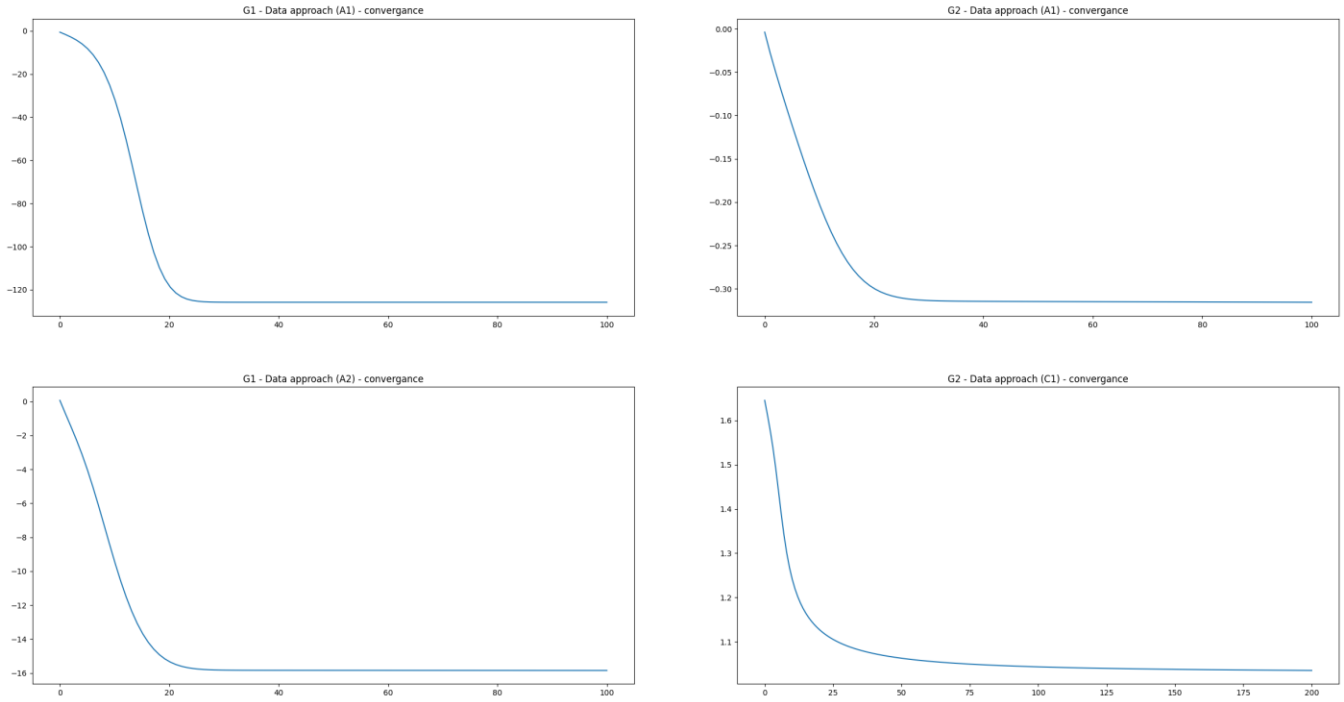
```
Numerical approach:
Slope 1 = 0.7943701494552656

Data approach:
Slope 1 (A1) = 0.8112979576427222
Slope 1 (A2) = 0.5748809798222166
```

Παρατήρηση: Κάποιες μέθοδοι προσεγγίσεως λειτουργούν πολύ καλύτερα από κάποιες άλλες.



Εικόνα 3: Αποτελέσματα νευρωνικών δικτύων για το πρόβλημα 3.1



Εικόνα 4: Σύγκλιση κόστους κατά τη διαδικασία εκπαίδευσης για το πρόβλημα 3.1

Πρόβλημα 3.2

Σε αυτό το πρόβλημα δίνεται το πρόβλημα αποφάσεως Markov:

$$S_{t+1} = \begin{cases} 0.8S_t + 1.0 + W_t, & a = 1 \\ -2.0 + W_t, & a = 2 \end{cases}, \qquad W_t = N(0,1)$$

Κάθε κατάσταση S δημιουργείται με βάση την προηγούμενη κατάσταση και την απόφαση α που πάρθηκε. Το κέρδος (reward) κάθε απόφασης υπολογίζεται από τον τύπο:

$$R(S_{t+1}) = min\left[2, {S_{t+1}}^2\right]$$

Ζητείται η επίλυση μέσω της μεθοδολογίας short sighted approach

Αριθμητική προσέγγιση

Μέσω της παραπάνω διαδικασίας δημιουργήθηκαν 1100 δείγματα καταστάσεων S με τυχαία ακολουθία για το α. Από αυτά χρησιμοποιήθηκαν τα πρώτα 500 (ζευγάρια S_t και S_{t+1}) της περίπτωσης α=1 και τα πρώτα 500 της περίπτωσης α=2. Εδώ το ζητούμενο είναι ο υπολογισμός των δύο δεσμευμένων πιθανοτήτων:

$$E_{S_{t+1}}{}^{a_1}[R(S_{t+1})|S_t] \quad και \quad E_{S_{t+1}}{}^{a_2}[R(S_{t+1})|S_t]$$

Οπότε ακολουθήθηκε η ίδια διαδικασία με το πρόβλημα 3.1 και υπολογίστηκαν οι δύο πίνακες $\mathcal{F}$, μέσω των οποίων σε συνδυασμό με τα διανύσματα R υπολογίστηκαν τα ζητούμενα διανύσματα $V_i = \mathcal{F}_i \cdot G_i$

Η δειγματοληψία και ο υπολογισμός αποτελεσμάτων υλοποιήθηκαν στο αρχείο **rlp_short_data.py**. Η παρουσίαση των αποτελεσμάτων και ο υπολογισμός της κλίσης της ευθείας $V_1(X)$ έγιναν στο αρχείο **results_2.py**.

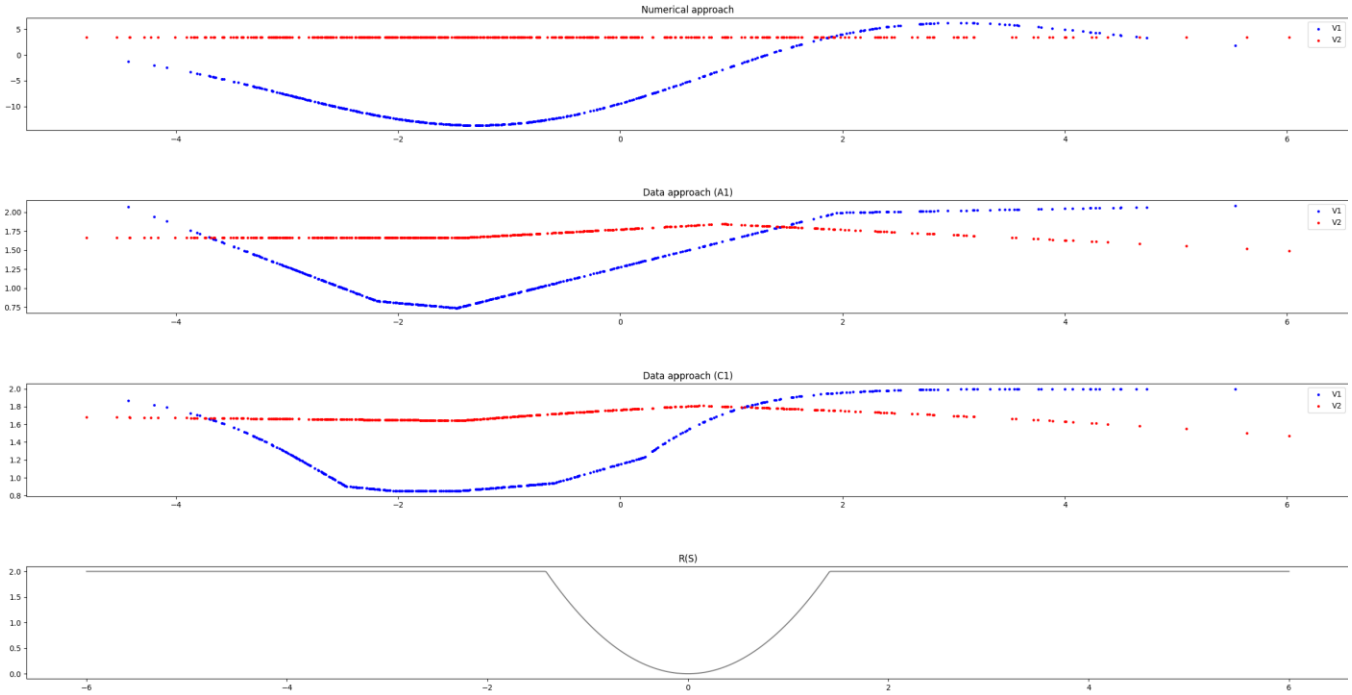
Προσέγγιση με νευρωνικά δίκτυα

Σε αυτή την προσέγγιση χρησιμοποιήθηκαν τα ίδια 1000 δείγματα. Καθώς το πρόβλημα είναι παρόμοιο με το πρόβλημα 3.1 η διαδικασία εκπαίδευσης των νευρωνικών δικτύων ήταν σχεδόν η ίδια. Οι διαφορές είναι ότι χρησιμοποιήθηκαν οι μέθοδοι A1, C1 και για τα δύο νευρωνικά δίκτυα, μόνο που η C1 είναι δεσμευμένη στο διάστημα [0, 2], οπότε αλλάζουν οι συναρτήσεις ω και φ για αυτή τη μέθοδο. Η τελευταία διαφορά είναι ότι χρησιμοποιήθηκε ο αλγόριθμος Stochastic Gradient Descent (SGD), ο οποίος για κάθε forward pass εφαρμόζει απευθείας αλλαγή παραμέτρων. Ο αλγόριθμος:

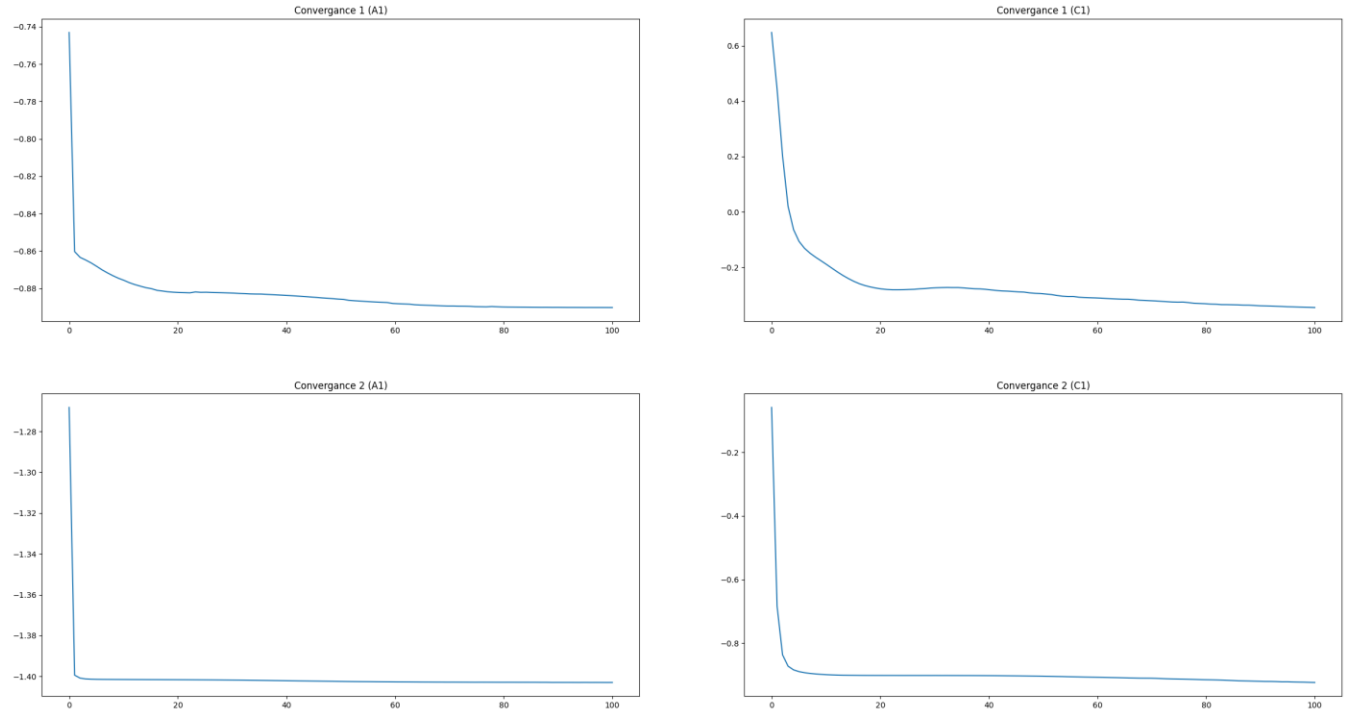
$$J(u) = \varphi\big(u(\mathcal{X})\big) + R(\mathcal{Y})\psi\big(u(\mathcal{X})\big)$$

$$\theta_t = \theta_{t-1} - \mu\big[R(Y_t) - \omega\big(u(X_t, \theta_{t-1})\big)\big]\rho\big(u(X_t, \theta_{t-1})\big)\nabla_\theta\big(u(X_t, \theta_{t-1})\big)$$

Ο παραπάνω αλγόριθμος επαναλήφθηκε 100 φορές για κάθε νευρωνικό δίκτυο με μ = 0.01. Η δημιουργία των νευρωνικών δικτύων και η εκπαίδευσή τους υλοποιήθηκαν στον αρχείο **rlp_short_data.py**. Η παρουσίαση των αποτελεσμάτων και ο υπολογισμός της κλίσης της ευθείας $V_1(X)$ έγιναν στο αρχείο **results_2.py**. Η υλοποίηση του νευρωνικού δικτύου βρίσκεται στο αρχείο **neural_net.py**.



Εικόνα 5: Αποτελέσματα αριθμιτικής μεθόδου και νευρωνικών δικτύων για το πρόβλημα 3.2



Εικόνα 6: Σύγκλιση κόστους κατά τη διαδικασία εκπαίδευσης για το πρόβλημα 3.2

Παρατηρώντας τις παραπάνω γραφικές παραστάσεις η βέλτιστη πολιτική αποφάσεων είναι:

- Απόφαση α_1 για $S = (-\infty, -3.5) \cup (1.75, +\infty)$
- Απόφαση α_2 για $S = [-3.5, 1.75]$

Πρόβλημα 3.3

Αυτό το πρόβλημα είναι το ίδιο με το 3.2

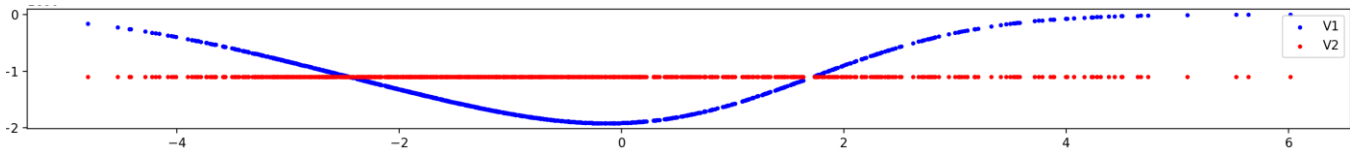
Αυτή τη φορά ζητείται η επίλυση του με τη χρήση της μεθοδολογίας infinite future reward.

Αριθμητική προσέγγιση

Αυτή τη φορά χρησιμοποιούνται και τα 1000 δείγματα ταυτόχρονα για τον υπολογισμό των διανυσμάτων V. Δημιουργήθηκαν οι δύο $\mathcal{F}$ πίνακες και υπολογίστηκαν τα διανύσματα V μέσω του επαναληπτικού τύπου:

$$V_j(S_{t+1}) = \mathcal{F}_j(R + \gamma \max\{v_1(S_t), v_2(S_t)\})$$

Οι αρχικές τιμές των διανυσμάτων είναι οι μηδενικές. Ο παραπάνω υπολογισμός επαναλήφθηκε 100 φορές στο αρχείο **rlp_numerical.py**.



Εικόνα 7: Προσέγγιση αριθμιτικής μεθόδου για πρόβλημα 3.3

Προσέγγιση με νευρωνικό δίκτυο

Σε αυτή τη προσέγγιση χρησιμοποιήθηκε ξανά ο αλγόριθμος SGD, αυτή τη φορά με τον παρακάτω τύπο:

$$\theta_t^j = \theta_{t-1}^j - \mu\Big[R(Y_t) + \gamma \max\{\omega\big(u(Y_t, \theta_{t-1}^1)\big), \omega\big(u(Y_t, \theta_{t-1}^2)\big)\Big\} - \omega\Big(u\big(Y_t, \theta_{t-1}^j\big)\Big)\Big]\rho\big(u(Y_t, \theta_{t-1}^1)\big)\nabla_\theta\big(u\big(X_i, \theta_{t-1}^j\big)\big)$$

Για αυτή την προσέγγιση χρησιμοποιήθηκαν ξανά οι μέθοδοι A1 και C1 μόνο που αυτή τη φορά η C1 είχε όρια [0, 10]. Η δημιουργία των νευρωνικών δικτύων και η εκπαίδευσή τους υλοποιήθηκαν στον αρχείο **rlp_inf_data.py**. Η παρουσίαση των αποτελεσμάτων και ο υπολογισμός της κλίσης της ευθείας $V_1(X)$ έγιναν στο αρχείο **results_3.py**. Η υλοποίηση του νευρωνικού δικτύου βρίσκεται στο αρχείο **neural_net.py**.

Cond_exp_data.py

```
from neural_net3 import NeuralNetwork
import numpy as np
import json

N = 500
n = 1
m = 50
k = 1
learning_rate = 0.001 # 0.001
epochs = 100 #100

# Generate samples
x_samples = np.random.normal(0, 20, N)
w_samples = np.random.normal(0, 1, N)
y_samples = 0.8 * x_samples + w_samples
g1_samples = y_samples
g2_samples = np.minimum(1, np.maximum(-1, y_samples))

# Create nns
nn1_A1 = NeuralNetwork(n, m, k, "A1")
nn1_A2 = NeuralNetwork(n, m, k, "A2")
nn2_A1 = NeuralNetwork(n, m, k, "A1")
nn2_C1 = NeuralNetwork(n, m, k, "C1_1")

# Training
output1_1 = nn1_A1.cond_exp_compute_gd(x_samples, g1_samples, learning_rate, epochs)
output1_2 = nn1_A2.cond_exp_compute_gd(x_samples, g1_samples, learning_rate, epochs)
output2_1 = nn2_A1.cond_exp_compute_gd(x_samples, g2_samples, learning_rate * 10,
epochs) # *10
output2_2 = nn2_C1.cond_exp_compute_gd(x_samples, g2_samples, learning_rate * 50,
epochs * 2) # *50, *2
results = np.array([output1_1[1], output1_2[1], output2_1[1], output2_2[1]])

# Store results
open("1_data_x.json", 'w').close()
open("1_data_1A1.json", 'w').close()
open("1_data_1A2.json", 'w').close()
open("1_data_2A1.json", 'w').close()
open("1_data_2C1.json", 'w').close()
open("1_data_res.json", 'w').close()

with open("1_data_x.json", 'w') as f: json.dump(x_samples.tolist(), f)
with open("1_data_1A1.json", 'w') as f: json.dump(output1_1[0].tolist(), f)
```

```
with open("1_data_1A2.json", 'w') as f: json.dump(output1_2[0].tolist(), f)
with open("1_data_2A1.json", 'w') as f: json.dump(output2_1[0].tolist(), f)
with open("1_data_2C1.json", 'w') as f: json.dump(output2_2[0].tolist(), f)
with open("1_data_res.json", 'w') as f: json.dump(results.tolist(), f)
```

cond_exp_numerical.py

```
import numpy as np
from scipy.stats import norm
import json

N = 500
xmin = -100
xmax = 100

def getf_matrix(y: np.ndarray, x: np.ndarray, N: int) -> np.ndarray:
    f_matrix = np.zeros(N**2).reshape(N, N)
    f_matrix[:, 0] = 0.5 * (norm.cdf(y[1] - 0.8 * x[:]) - norm.cdf(y[0] - 0.8 * x[:]))
    f_matrix[:, N-1] = 0.5 * (norm.cdf(y[N-1] - 0.8 * x[:]) - norm.cdf(y[N-2] - 0.8 *
x[:]))
    for i in range(N):
        f_matrix[i, 1:N-2] = 0.5 * (norm.cdf(y[2:N-1] - 0.8 * x[i]) - norm.cdf(y[0:N-3]
- 0.8 * x[i]))
    return f_matrix

x_samples = np.linspace(xmin, xmax, N)
w_samples = np.random.normal(0, 1, N)
y_samples = 0.8 * x_samples + w_samples

g1_vector = y_samples.reshape(N, 1)
g2_vector = np.minimum(1, np.maximum(-1, y_samples)).reshape(N, 1)
f_matrix = getf_matrix(y_samples, x_samples, N)
v1_vector = np.array(np.dot(f_matrix, g1_vector)).reshape(N)
v2_vector = np.array(np.dot(f_matrix, g2_vector)).reshape(N)

open("1_numerical_x.json", 'w').close()
open("1_numerical_1.json", 'w').close()
open("1_numerical_2.json", 'w').close()

with open("1_numerical_x.json", 'w') as f: json.dump(x_samples.tolist(), f)
with open("1_numerical_1.json", 'w') as f: json.dump(v1_vector.tolist(), f)
with open("1_numerical_2.json", 'w') as f: json.dump(v2_vector.tolist(), f)
```

neural_net.py

```python
import numpy as np

nn_method = enumerate(("A1", "A2", "C1_1", "C1_2", None))

def ReLU(x: np.ndarray) -> np.ndarray:
    return np.maximum(0, x)

def der_ReLU(x: np.ndarray) -> np.ndarray:
    return np.where(x > 0, 1, 0)

def rho(z: np.ndarray, method: enumerate) -> np.ndarray:
    match method:
        case "A1":
            return -np.ones(len(z))
        case "A2":
            return -np.exp(-0.5 * np.abs(z))
        case "C1_1" | "C1_2" | "C1_3":
            return -np.exp(z) / (1 + np.exp(z))

def omega(z: np.ndarray, method: enumerate) -> np.ndarray:
    match method:
        case "A1":
            return z
        case "A2":
            return np.sinh(z)
        case "C1_1":
            return (np.exp(z) - 1) / (1 + np.exp(z))
        case "C1_2":
            return 2 * np.exp(z) / (1 + np.exp(z))
        case "C1_3":
            return 10 * np.exp(z) / (1 + np.exp(z))

def phi(z: np.ndarray, method: enumerate) -> np.ndarray:
    match method:
        case "A1":
            return 0.5 * np.square(z)
        case "A2":
            return np.exp(0.5 * np.abs(z)) + np.exp(-1.5 * np.abs(z)) / 3 - 4 / 3
        case "C1_1" | "C1_2":
            return 2 / (1 + np.exp(z)) + np.log(1 + np.exp(z))
        case "C1_3":
            return 10 / (1 + np.exp(z)) + 10 * np.log(1 + np.exp(z))
```

```python
def psi(z: np.ndarray, method: enumerate) -> np.ndarray:
    match method:
        case "A1":
            return -z
        case "A2":
            return 2 * np.sign(z) * (np.exp(-0.5 * np.abs(z)) - 1)
        case "C1_1" | "C1_2" | "C1_3":
            return -np.log(1 + np.exp(z))

class NeuralNetwork:
    def __init__(self, in_size: int, hid_size: int, out_size: int, method: enumerate,
SGD=False):
        self.n = in_size
        self.m = hid_size
        self.k = out_size
        self.method = method
        self.SGD = SGD

        # Weights and offsets
        self.A1 = np.random.normal(0, 1/(self.n + self.m), (self.n, self.m)) # n x m
        self.A2 = np.random.normal(0, 1/(self.m + self.k), (self.m, self.k)) # m x k
        self.B1 = np.zeros((1, self.m)) # 1 x m
        self.B2 = np.zeros((1, self.k)) # 1 x k

        if not self.SGD:
            self.sdA1 = np.array([])
            self.sdB1 = np.array([])
            self.sdA2 = np.array([])
            self.sdB2 = np.array([])
            self.Z_storage = np.array([])

        self.cost = np.array([])

    def forward(self, X: np.ndarray):
        # W1 = A1 x X + B1 -- 1 x m
        self.W1 = np.array(np.dot(X, self.A1) + self.B1)
        # Z1 = ReLU(W1) -- 1 x m
        self.Z1 = ReLU(self.W1)
        # Z = W2 = A2 x Z1 + B2 -- 1 x k
        self.Z = np.array(np.dot(self.Z1, self.A2) + self.B2)

        if not self.SGD: self.Z_storage = np.append(self.Z_storage, self.Z)

    def calc_cost(self, DataModel_output: float):
        g = DataModel_output
```

```
        z = self.Z
        self.J = phi(z, self.method) + g * psi(z, self.method)

        self.cost = np.append(self.cost, self.J)

    def backward(self, X: np.ndarray):
        u2 = 1
        v2 = u2
        u1 = v2 * self.A2.T
        v1 = u1 * der_ReLU(self.W1)

        self.dA2 = np.dot(self.Z1.T, v2)
        self.dB2 = v2
        self.dA1 = np.dot(X.T, v1)
        self.dB1 = v1

        if not self.SGD:
            self.sdA1 = np.append(self.sdA1, self.dA1)
            self.sdB1 = np.append(self.sdB1, self.dB1)
            self.sdA2 = np.append(self.sdA2, self.dA2)
            self.sdB2 = np.append(self.sdB2, self.dB2)

    def updateParams_gd(self, g_data: np.ndarray, learning_rate: float):
        z = self.Z_storage

        self.sdA1 = self.sdA1.reshape(len(z), self.n, self.m)
        self.sdB1 = self.sdB1.reshape(len(z), 1, self.m)
        self.sdA2 = self.sdA2.reshape(len(z), self.m, self.k)
        self.sdB2 = self.sdB2.reshape(len(z), 1, self.k)

        grad_delta = (g_data - omega(z, self.method)) * rho(z, self.method)
        gradA1 = np.zeros_like(self.sdA1)
        gradB1 = np.zeros_like(self.sdB1)
        gradA2 = np.zeros_like(self.sdA2)
        gradB2 = np.zeros_like(self.sdB2)

        gradA1 = grad_delta[:, np.newaxis, np.newaxis] * self.sdA1
        gradB1 = grad_delta[:, np.newaxis, np.newaxis] * self.sdB1
        gradA2 = grad_delta[:, np.newaxis, np.newaxis] * self.sdA2
        gradB2 = grad_delta[:, np.newaxis, np.newaxis] * self.sdB2

        self.A1 = self.A1 - learning_rate * np.average(gradA1, axis=0)
        self.B1 = self.B1 - learning_rate * np.average(gradB1, axis=0)
        self.A2 = self.A2 - learning_rate * np.average(gradA2, axis=0)
        self.B2 = self.B2 - learning_rate * np.average(gradB2, axis=0)
```

```
    def updateParams_sgd(self, reward: float, learning_rate: float):
        z = self.Z

        grad_delta = (reward - omega(z, self.method)) * rho(z, self.method)
        self.A1 = self.A1 - learning_rate * grad_delta * self.dA1
        self.B1 = self.B1 - learning_rate * grad_delta * self.dB1
        self.A2 = self.A2 - learning_rate * grad_delta * self.dA2
        self.B2 = self.B2 - learning_rate * grad_delta * self.dB2

    def updateParams_sgd_inf(self, reward: float, z_curr: float, z_foreign: float,
learning_rate: float):
        z_next = self.Z

        grad_delta = (reward + 0.8 * np.max(omega(np.array([z_next, z_foreign]),
self.method)) - omega(z_curr, self.method)) * rho(z_curr, self.method)
        self.A1 = self.A1 - learning_rate * grad_delta * self.dA1
        self.B1 = self.B1 - learning_rate * grad_delta * self.dB1
        self.A2 = self.A2 - learning_rate * grad_delta * self.dA2
        self.B2 = self.B2 - learning_rate * grad_delta * self.dB2

    def empty_arrays(self):
        self.sdA1 = np.array([])
        self.sdB1 = np.array([])
        self.sdA2 = np.array([])
        self.sdB2 = np.array([])
        self.Z_storage = np.array([])
        self.cost = np.array([])

    def cond_exp_compute_gd(self, X_dataset: np.ndarray, DataModel_output: np.ndarray,
learning_rate: float, epochs: int) -> list[np.ndarray, np.ndarray]:
        convergance = np.array([])

        for _ in range(epochs):
            for j in range(len(X_dataset)):
                self.forward(X_dataset[j])
                self.calc_cost(DataModel_output[j])
                self.backward(X_dataset[j])

            self.updateParams_gd(DataModel_output, learning_rate)
            convergance = np.append(convergance, np.average(self.cost))
            self.empty_arrays()

        for i in range(len(X_dataset)):
            self.forward(X_dataset[i])
```

```
        return [convergance, omega(self.Z_storage, self.method)]

    def cond_exp_compute_sgd(self, X_dataset: np.ndarray, DataModel_output: np.ndarray,
learning_rate: float, epochs: int) -> list[np.ndarray, np.ndarray]:
        output = np.array([])
        convergance = np.array([])

        for _ in range(epochs):
            for j in range(len(X_dataset)):
                self.forward(X_dataset[j])
                self.calc_cost(DataModel_output[j])
                self.backward(X_dataset[j])
                self.updateParams_sgd(DataModel_output[j], learning_rate)

            convergance = np.append(convergance, np.average(self.cost))
            self.empty_arrays()

        for i in range(len(X_dataset)):
            self.forward(X_dataset[i])
            output = np.append(output, self.Z)

        return [convergance, omega(output, self.method)]
```

results_1.py

```
import numpy as np
from matplotlib import pyplot as plt
import json

# Load results from numerical method
with open("1_numerical_x.json", 'r') as f: num_x = np.array(json.load(f))
with open("1_numerical_1.json", 'r') as f: num_1 = np.array(json.load(f))
with open("1_numerical_2.json", 'r') as f: num_2 = np.array(json.load(f))

# Load results from data method
with open("1_data_x.json", 'r') as f: data_x = np.array(json.load(f))
with open("1_data_res.json", 'r') as f: data_approach = np.array(json.load(f))
data_1A1 = data_approach[0]
data_1A2 = data_approach[1]
data_2A1 = data_approach[2]
data_2C1 = data_approach[3]
with open("1_data_1A1.json", 'r') as f: data_conv_1A1 = np.array(json.load(f))
with open("1_data_1A2.json", 'r') as f: data_conv_1A2 = np.array(json.load(f))
```

```
with open("1_data_2A1.json", 'r') as f: data_conv_2A1 = np.array(json.load(f))
with open("1_data_2C1.json", 'r') as f: data_conv_2C1 = np.array(json.load(f))

# Present the slope results
slope_num1 = np.average(num_1 / num_x)
slope_num2 = np.average(num_2 / num_x)
print("\nNumerical approach:\nSlope 1 =", slope_num1)
slope_data1_A1 = np.average(data_1A1 / data_x)
slope_data1_A2 = np.average(data_1A2 / data_x)
slope_data2_A1 = np.average(data_2A1 / data_x)
slope_data2_C1 = np.average(data_2C1 / data_x)
print("\nData approach:\nSlope 1 (A1) =", slope_data1_A1, "\nSlope 1 (A2) =",
slope_data1_A2)

# Plot the results
x1 = np.linspace(0, len(data_conv_1A1), len(data_conv_1A1))
x2 = np.linspace(0, len(data_conv_2C1), len(data_conv_2C1))
figure, axis = plt.subplots(3, 2)

axis[0, 0].plot(num_x, num_1)
axis[0, 0].set_title("G1 = Y - Numerical approach")

axis[0, 1].plot(num_x, num_2)
axis[0, 1].set_title("G2 = min(1, max(-1, Y)) - Numerical approach")

axis[1, 0].scatter(data_x, data_1A1, s=2, linewidths=2)
axis[1, 0].set_title("G1 - Data approach (A1)")

axis[2, 0].scatter(data_x, data_1A2, s=2, linewidths=2)
axis[2, 0].set_title("G1 - Data approach (A2)")

axis[1, 1].scatter(data_x, data_2A1, s=2, linewidths=2)
axis[1, 1].set_title("G2 - Data approach (A1)")

axis[2, 1].scatter(data_x, data_2C1, s=2, linewidths=2)
axis[2, 1].set_title("G2 - Data approach (C1)")

plt.tight_layout(pad=0, h_pad=0, w_pad=0)
plt.show()

figure, axis = plt.subplots(2, 2)

axis[0, 0].plot(x1, data_conv_1A1)
axis[0, 0].set_title("G1 - Data approach (A1) - convergance")
```

```
axis[1, 0].plot(x1, data_conv_1A2)
axis[1, 0].set_title("G1 - Data approach (A2) - convergance")

axis[0, 1].plot(x1, data_conv_2A1)
axis[0, 1].set_title("G2 - Data approach (A1) - convergance")

axis[1, 1].plot(x2, data_conv_2C1)
axis[1, 1].set_title("G2 - Data approach (C1) - convergance")

plt.tight_layout(pad=0, h_pad=0, w_pad=0)
plt.show()
```

results_2.py