Data-Driven Techniques: 3rd Homework

Μαρία-Νίκη Ζωγράφου

AM: 1096060

Περιεχόμενα

Ερώτημα 1

1α)

Γνωστά ότι: $Y = 0.8X + W, W \sim N(0,1)$, επομένως:

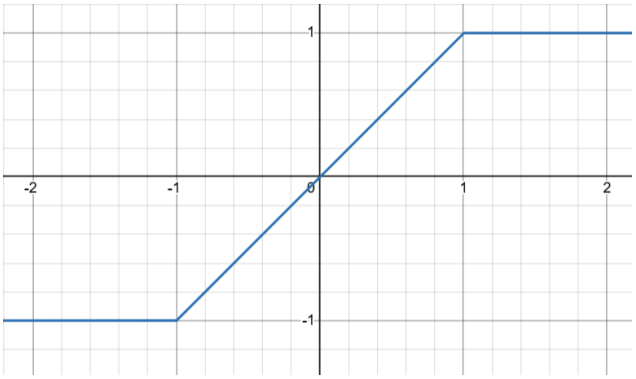
$$E[Y|X = x] = E[(0.8X + W)|X = x] = E[0.8X|X = x] + 0 = 0.8x$$
$$και$$
$$E[\min(1, \max(-1, Y))\,|X = x]$$

Η συνάρτηση $\min(1, \max(-1, Y))$ *είναι η:*

$$g(y) = \begin{cases} sign(y) * 1, |y| > 1 \\ y, -1 < y < 1 \end{cases}$$

Δηλαδή:

$$g(y) = \begin{cases} 1, & y > 1 \\ y, & -1 < y < 1 \\ -1, & y < -1 \end{cases}$$



$$E[g(Y)|X = x] = \int_{-\infty}^{-1} -1 * f_{Y|X}\,(y|x)dy + \int_{-1}^{1} y * f_{Y|X}\,(y|x)dy + \int_{1}^{\infty} 1 * f_{Y|X}\,(y|x)dy$$

Αριθμητική μέθοδος υπολογισμού E[G(Y)|X=x]:

Χωρίζουμε πεδίου ορισμού σε μικρά διαστήματα Ai. Θεωρούμε P(Y ∈ Aj | X = x) σταθερή σε κάθε διάστημα Aj. Εκτιμάμε την τιμή της συνάρτησης G(Y) με sampling.

$E[G(Y)|X = x] = u(x) = \sum_j G(y_i) * P(Y \in A_j\,|X = x)$, *όπου* y_j *το δείγμα απο* A_j, δηλαδή $u(x) = \int_{Ay} G(y) * h(y|x)dy$, *όπου* $h(y|x)$ *η* $pdf(probability\ density\ function)$.

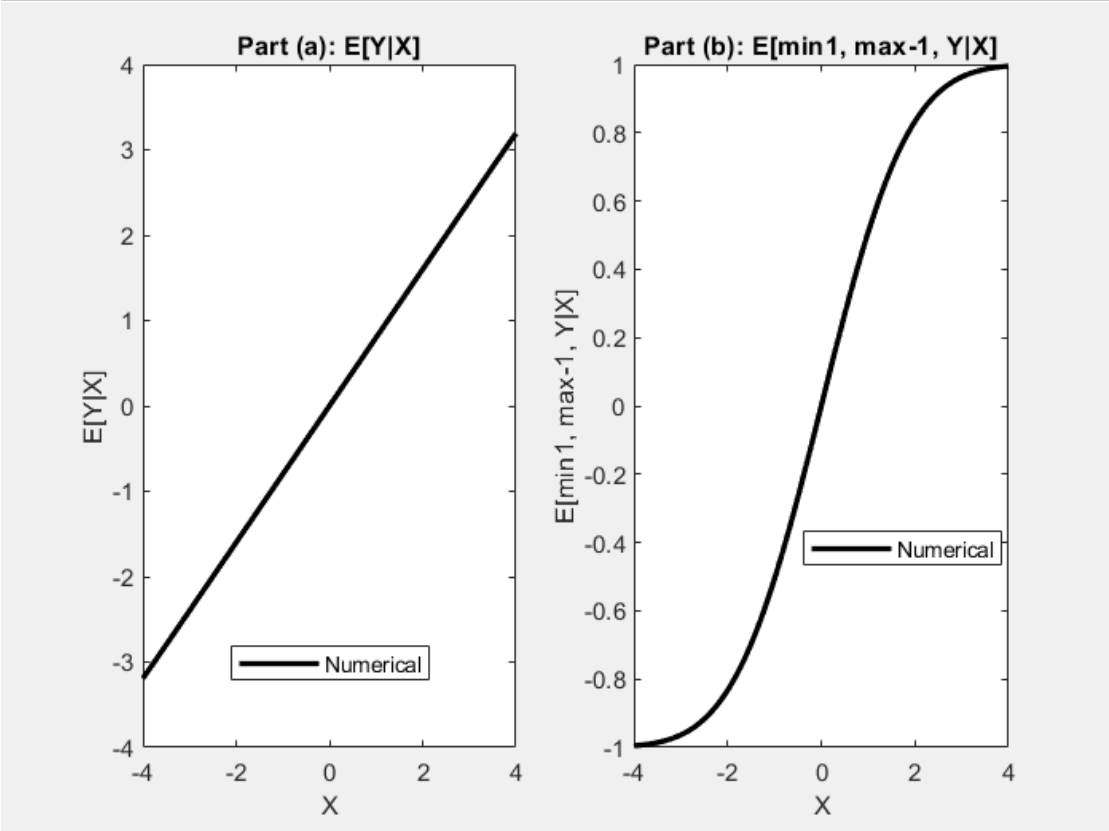
Η ολοκλήρωση γίνεται με την μέθοδο των τραπεζίων ($u_t(x) = \frac{G(y_1)+G(y_0)}{2} * (H(y_1|x) - H(y_0|x) + \cdots)$, *όπου* H *η* cdf) την οποία αναπαριστούμε σε μορφή πινάκων. Καταλήγουμε σε:

$$U = F * G$$

Η cdf στην συγκεκριμένη περίπτωση είναι η $\Phi\left(\frac{z-\mu}{\sigma}\right)$ = `normcdf((y - 0.8 * x), 0, 1)` και έτσι θα υπολογίσουμε τις τιμές του F matrix.

Για τις τιμές του G matrix θα χρησιμοποιήσουμε την εκάστοτε συνάρτηση, δηλαδή για E[Y|X] έχουμε G(y) = y και για E[min{1, max{-1, Y}}|X] έχουμε G(y)= min{1, max{-1, Y}}.

Λαμβάνουμε τα παρακάτω αποτελέσματα:



1β)

Θα επιχειρήσουμε να επιλύσουμε το παραπάνω πρόβλημα με χρήση νευρωνικών δικτύων, χρησιμοποιώντας ζεύγη X,Y για εκπαίδευση.

Input: X

Output: G(Y): Y ή min(1, max(-1,Y))

Αρχιτεκτονική Νευρωνικού: 1x50x500

Loss Function: αναλόγως

Για την εκτίμηση του E[Y|X=x]:

Εκφώνηση: «Note that the first conditional expectation is unbounded therefore provide a data-driven estimate using [A1],[A2] ».

- [A1]: w(z)=z, ρ(z)=-1, ϕ(z)=z^2/2, ψ(z)=-z, Range R real line
- [A2]: w(z)=sinh(z), ρ(z)=-exp(-0.5*|z|), ϕ(z)=(exp(0.5*|z|)-1)+1/3*(exp(-1.5*|z|)-1), ψ(z)=2. * sign(z)*(exp(-0.5*|z|)-1)

Έχουμε loss $\boldsymbol{J(\theta)} = \frac{\mathbf{1}}{\boldsymbol{n}}\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{\varphi}(\boldsymbol{u}(\boldsymbol{x_i}, \boldsymbol{\theta})) + \boldsymbol{G}(\boldsymbol{y_i})\boldsymbol{\psi}\big(\boldsymbol{u}(\boldsymbol{x_i}, \boldsymbol{\theta})\big)$και θέλουμε $\theta_0 = argminJ(\theta)$ ώστε u(x,θ) ≈ u_0(x), **<u>όπου G θα είναι τα δείγματα Y</u>**.

Θα χρησιμοποιήσουμε gradient descent με learning rate μ:

$$\theta_t = \theta_{t-1} - \frac{\mu}{n}\sum_{i=1}^{n}\big[G(y_i) - w\big(u(x_i, \theta_t - 1\,)\big)\big]\rho\big(u(x_i, \theta_t - 1\,)\big)\nabla_\theta u(x_i, \theta_t - 1\,)$$

$$\theta_t = \theta_{t-1} - \frac{\mu}{n}\sum_{i=1}^{n} \nabla_\theta\{\varphi\big(u(x_i, \theta_{t-1})\big) + G(y_i)\psi(u(x_i, \theta_{t-1}\,)$$

$$\theta_t = \theta_{t-1} - \frac{\mu}{n}\nabla_\theta \boldsymbol{J}(\boldsymbol{\theta})$$

Η σχέση είναι από το Paper της Lecture 11 (Στο paper: Note that we have absorbed in μ the division by the constant n)

$$\theta_t = \theta_{t-1} - \mu \sum_{i=1}^{n} \nabla_\theta \mathrm{h}(X_i, \theta_{t-1}) \qquad (13)$$

Νευρωνικό δίκτυο:

Από είσοδο στο κρυφό επίπεδο: Z1 = W1 * X_samples + b1

Activation function: A1 = relu(Z1)

Από κρυφό επίπεδο στην έξοδο: Z2 = W2* A1 + b2

Τελικό αποτέλεσμα: u_X = Z2

Gradient Descent:

Επομένως πρέπει να υπολογίσουμε τις παραγώγους κάθε θ.

GRADIENT DESCENT για A1

$$[\mathrm{A1}]: \mathrm{w(z)} = \mathrm{z}, \rho(\mathrm{z}) = -1, \varphi(\mathrm{z}) = \frac{\mathrm{z}^2}{2}, \psi(\mathrm{z}) = -\mathrm{z}$$

Για A1 έχουμε: $\boldsymbol{J} = \frac{\mathbf{1}}{\boldsymbol{N}}\sum \frac{\boldsymbol{u(x,\theta)^2}}{\mathbf{2}} + \boldsymbol{G} * \big(-\boldsymbol{u(x,\theta)}\big)\ \boldsymbol{\Sigma\chi}.\,(\boldsymbol{A1})$

Ξεκινάμε από το Z2: $\frac{\partial J}{\partial Z2} = \frac{\partial J}{\partial u(x,\theta)} \overset{\Sigma\chi.(A1)}{\Longrightarrow} \frac{\partial J}{\partial u(x,\theta)} = \boldsymbol{u(x,\theta) - Y = dZ2}$ ενώ $u(x,\theta) = Z$

Αφού Z2=W2*A1+b2, θα χρησιμοποιήσουμε τον κανόνα της αλυσίδας:

$\frac{\partial J}{\partial Z2}\frac{\partial Z2}{\partial W2} = \frac{\partial J}{\partial W2}$ ενώ $\frac{\partial Z2}{\partial W2} = A1$. Δηλαδή $\frac{\partial J}{\partial W2} = \frac{1}{n}\sum_1^n (u(x,\theta) - Y_i) * A1 = \frac{1}{n} dZ2 * A1^T$

Επομένως $\boldsymbol{W2_t = W2_{t-1} - \frac{\mu}{N} * A1^T * (u(x,\theta) - Y)}$, $\delta\eta\lambda.\ \boldsymbol{dW2 = \frac{\mu}{N} * A1^T * (u(x,\theta) - Y)}$.

$\frac{\partial J}{\partial Z2}\frac{\partial Z2}{\partial b2} = \frac{\partial J}{\partial b2}$ ενώ $\frac{\partial Z2}{\partial b2} = 1$. Επομένως $\boldsymbol{b2_t = b2_{t-1} - \frac{\mu}{N} * \sum(u(x,\theta) - Y)}$, δηλαδή:

$$\mathbf{db2} = \frac{\boldsymbol{\mu}}{\boldsymbol{N}} * \boldsymbol{\Sigma dZ2}$$

Στη συνέχεια υπολογίζουμε ότι: $\frac{\partial J}{\partial Z2}\frac{\partial Z2}{\partial A1} = \frac{\partial J}{\partial A1} = dZ2 * W2^T$, $\boldsymbol{A1_t = A1_{t-1} - W2^T * dZ2}$

Θα χρειαστούμε την παράγωγο της ReLU, $relu' = \begin{cases} 1, z > 0 \\ 0, z < 0 \end{cases}$.

Με την ίδια λογική έχουμε $\mathbf{dZ1} = \boldsymbol{relu'} * \boldsymbol{W2^T} * \boldsymbol{dZ2}$, $\mathbf{dW1} = \frac{\mathbf{1}}{\mathbf{N}}\boldsymbol{dZ1} * \boldsymbol{X^T}$,

$$\boldsymbol{db1} = \frac{\mathbf{1}}{\boldsymbol{N}}\boldsymbol{\Sigma dZ1}$$

GRADIENT DESCENT για A2

Ακολουθούμε την ίδια διαδικασία αλλά για τις συναρτήσεις A2. Τα dW2, db2, dA1, dZ1, dW1, db1 τηρούν τις ίδιες σχέσεις με προηγουμένως, διότι έχουμε την ίδια αρχιτεκτονική νευρωνικού δικτύου. Όμως το u και το dZ2 θα αλλάξουν, καθώς το Loss function είναι διαφορετικό, αφού χρησιμοποιούμε διαφορετικές φ και ψ συναρτήσεις:

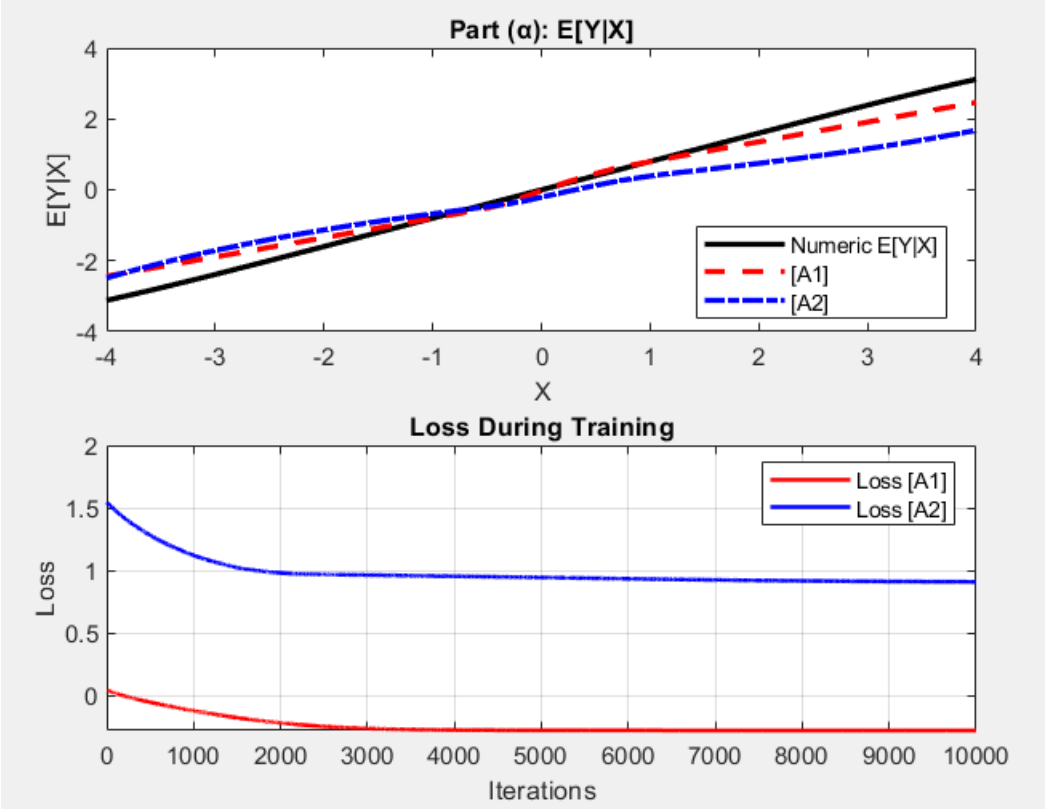
$$\text{Loss J} = \frac{1}{\text{n}}\sum_{1}^{n}(\varphi\left(u(x_i,\theta) + G(Y) * \psi\big(u(x_i,\theta)\big)\right)$$

$$[A2]: \varphi(z) = \left(e^{(0.5|z|)} - 1\right) + \frac{1}{3}\left(e^{(-1.5|z|)} - 1\right), \psi(z) = 2 * sign(z) * \left(e^{(-0.5|z|)} - 1\right),$$

$\omega(\text{z}) = \sinh(\text{z})$. *H u είναι μετασχηματισμός του Z2* $\boldsymbol{\kappa\alpha\iota}\ \boldsymbol{u(X,\theta)} = -\boldsymbol{e}^{\frac{|z_2|}{2}}\left(\boldsymbol{Y} - \boldsymbol{sinh}(\boldsymbol{Z_2})\right)$

Ενώ $\boldsymbol{dZ_2} = sign(u_x)(0.5 * \left(e^{0.5|u_x|} - e^{-1.5|u_x|}\right) - Y * e^{-0.5|u_x|})$

Αποτελέσματα



Οι απώλειες συγκλίνουν, ενώ και τα δύο νευρωνικά δίκτυα προσεγγίζουν πολύ καλά την αριθμητική μέθοδο. Τα δυο δίκτυα έχουν σχεδόν την ίδια συμπεριφορά. Παρατηρούμε όμως ότι με τις συναρτήσεις A2 η σύγκλιση γίνεται γρηγορότερα, ταυτόχρονα όμως το loss παραμένει μεγαλύτερο. Το A1 κάνει καλύτερη προσέγγιση ωστόσο.

Χρησιμοποιήθηκε: `num_iterations = 10000, learning rate = 0.001.`

Για την εκτίμηση του E[[min{1,max{−1,Y}}|X=x]:

Χρησιμοποιούμε την ίδια μεθοδολογία με πριν, κάνοντας δυο νευρωνικά δίκτυα με την προηγούμενη αρχιτεκτονική. Στο πρώτο χρησιμοποιούμε τις συναρτήσεις A1 και στο δεύτερο τις C1 για bounded σε [-1,1] (α=-1,β=1). Για το C1 χρειάζεται να υπολογίσουμε τις **u** και **dZ_2**.

$$[C1]: \varphi(z) = \frac{2}{1+e^z} + log(1+e^z), \psi(z) = -log(1+e^z), \omega(z) = (\frac{-1}{1+e^z}) + (\frac{e^z}{1+e^z})$$

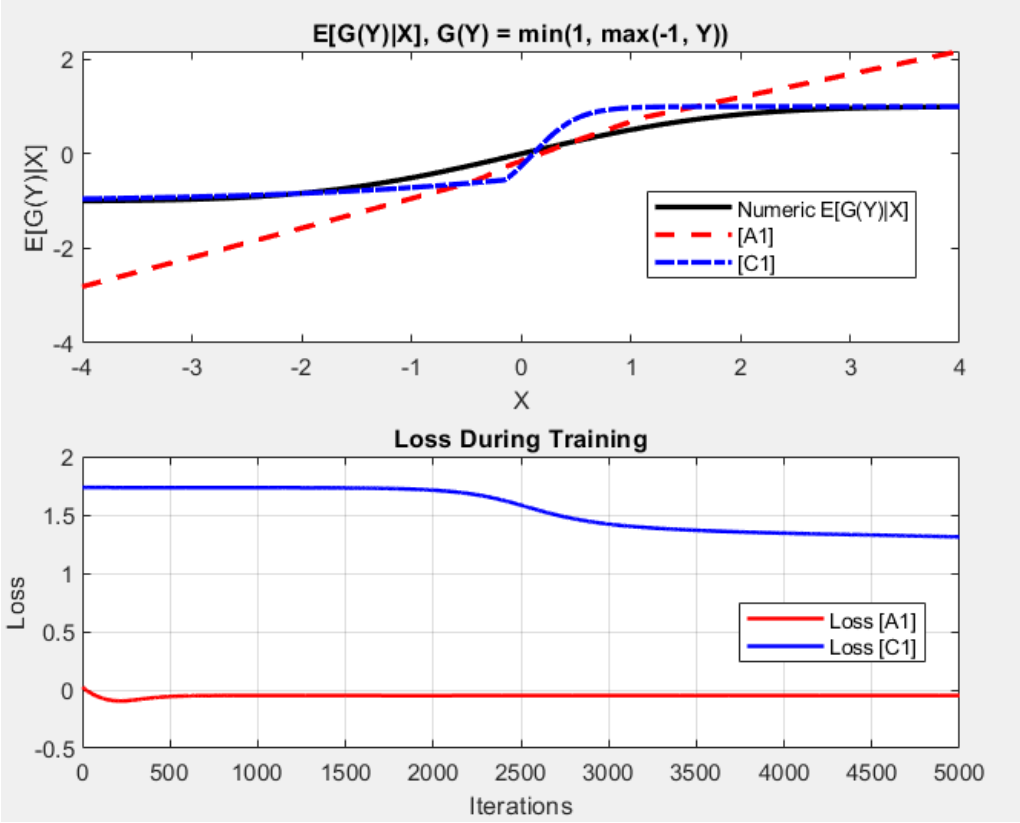
$$\mathrm{u} = -(\frac{1}{1+e^{z_2}}) * (\mathrm{Y} - \frac{1}{1+e^{z_2}} - \frac{-1}{1+e^{z_2}})$$

$$dZ2 = -(2 * \frac{e^{Z2}}{(1+(e^{Z2})^2)} + (\frac{e^{Z2}}{(1+e^{Z2})}) - Y * (\frac{e^{Z2}}{(1+e^{Z2})})$$

Απόδειξη

First prove that EY[G(Y)|X = X] is bounded in [−1,1]: όπως αναλύθηκε στο ερώτημα 1^{α}, η g(y) είναι φραγμένη στο [-1,1].

Αποτελέσματα



Οι απώλειες συγκλίνουν. Το νευρωνικό δίκτυο C1 προσεγγίζει πολύ καλά την αριθμητική μέθοδο ενώ το A1 αποκλίνει, κάνοντας ωστόσο μια αρκετά καλή προσέγγιση κοντά στο 0.

Παρατηρούμε όμως ότι με τις συναρτήσεις C1 η σύγκλιση γίνεται αργότερα και το loss παραμένει μεγαλύτερο. Είναι όμως πιο αποτελεσματικό από το A1.

Χρησιμοποιήθηκε: `num_iterations = 3000, learning rate = 0.01.`

Προσοχή:

Σχεδιάζουμε την ω(u(x,θ)), καθώς από διάλεξη 12:

if $\mathrm{u_o}(X) = \arg\min_{\mathrm{u}} J(\mathrm{u})$, then: $\omega(\mathrm{u_o}(X)) = \mathbb{E}_y[\mathrm{g}(y)|X = X]$

Ερώτημα 2

Αριθμητική Προσέγγιση:

Έχουμε μια Μαρκοβιανή Decision Process με α = action και W~$N(0,1)$ Gaussian Noise:

$\{Για\ a = 1\colon\ \ S_{t+1} = S_t + 1 + W\ ,\qquad για\ a = 2\colon\ \ S_{t+1} = -2 + W\}$

Υπολογίζουμε τις cdf ώστε να εφαρμόσουμε την αριθμητική μέθοδο. Ο όρος S_t+1 και ο όρος -2 είναι ντετερμινιστικοί ενώ το W ακολουθεί κανονική κατανομή, επομένως και για τις δυο περιπτώσεις κάνουμε απλώς έναν γραμμικό μετασχηματισμό στην κανονική κατανομή.

Για action α=1:

$E[S_{t+1}] = E[0.8S_t + 1 + W] = 0.8S_t + 1 + E[W] = 0.8S_t + 1$ και

$Var[S_{t+1}] = Var[0.8S_t + 1 + W] = Var(W) = 1.$

Επομένως: S_{t+1}~$N(0.8S_t + 1,1)$, άρα $\mathrm{H}_1(\mathrm{Y} \mid \mathrm{X}) = \Phi(\frac{\mathrm{Y}-(0.8S_t+1)}{1})$

Για action α=2:

$E[S_{t+1}] = E[-2 + W] = -2 + E[W] = -2$ και $Var[S_{t+1}] = Var(W) = 1.$

Επομένως: S_{t+1}~$N(-2,1)$, άρα $\mathrm{H}_2(\mathrm{Y} \mid \mathrm{X}) = \Phi(\frac{\mathrm{Y}+2}{1})$

Μήτρες F και G:

Τον ρόλο του Y παίρνει η S_{t+1} και του X η S_t . Όπως και προηγουμένως υπολογίζουμε την μήτρα F με τους τύπους της διάλεξης 12, μόνο που τώρα έχουμε δύο μήτρες ***F1*** (χρήση H1) και ***F2*** (χρήση H2).

and matrix $\mathscr{F}$ of dimensions $m \times n$ with elements

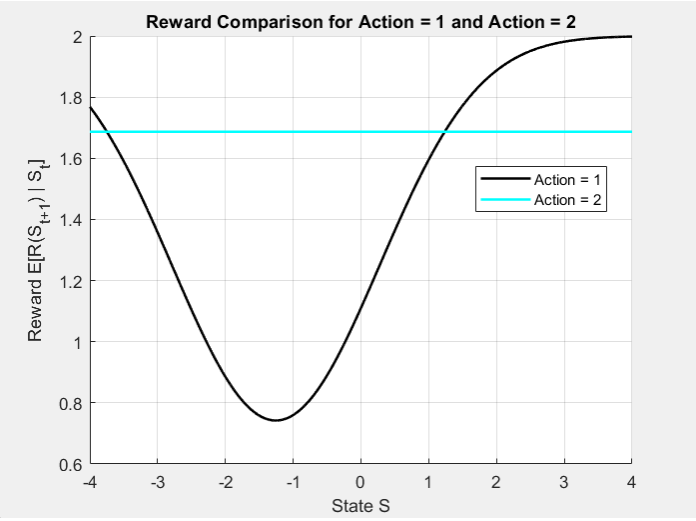
$$\mathscr{F}_{j1} = 0.5\big(\mathrm{H}(y_1|x_j) - \mathrm{H}(y_0|x_j)\big),\quad \mathscr{F}_{jn} = 0.5\big(\mathrm{H}(y_n|x_j) - \mathrm{H}(y_{n-1}|x_j)\big)$$
$$\mathscr{F}_{ji} = 0.5\big(\mathrm{H}(y_i|x_j) - \mathrm{H}(y_{i-2}|x_j)\big),\ i = 2, \ldots, n-1$$

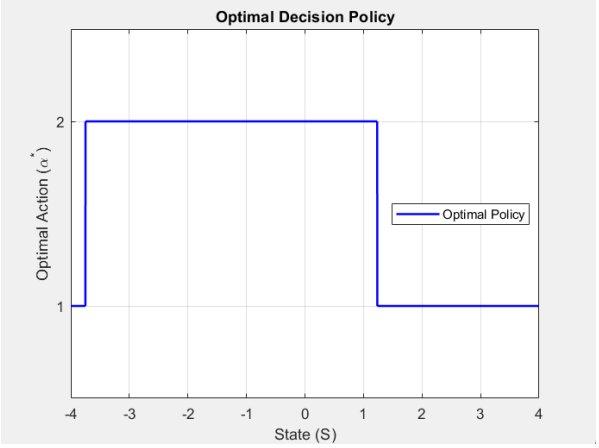
then approximation is matrix/vector product $\mathrm{V} \approx \mathscr{F}\mathrm{G}$

Για μήτρα ***G*** θα χρησιμοποιήσουμε τα Rewards $R = \min\{2, S^2\}$, που δίνονται στα S_{t+1}.

Τέλος θα υπολογίζουμε το Optimal policy ως εξής: **a = arg max {v1,v2}.**

Αποτελέσματα:





Νευρωνικά Δίκτυα

Θα χρειαστούμε δύο είδη Νευρωνικών δικτύων ένα που χρησιμοποιεί τις [A1] και ένα τις [C1] συναρτήσεις (βρισκόμαστε στο [0,2] άρα a=0, b=2 από το Moustakides 2024 paper).

$$[\mathrm{A1}]: \mathrm{w(z)} = \mathrm{z}, \rho(\mathrm{z}) = -1, \varphi(\mathrm{z}) = \frac{\mathrm{z}^2}{2}, \psi(\mathrm{z}) = -\mathrm{z}$$

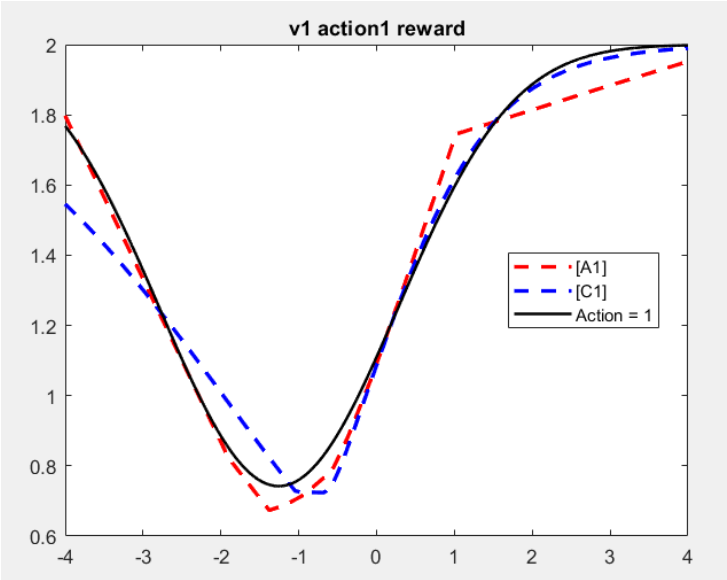
$$[\mathrm{C1}]: \omega(\mathrm{z}) = 2 * \frac{e^z}{1+e^z}, \varphi(z) = \frac{2}{1+e^z}, \psi(z) = -log(1+e^z)$$

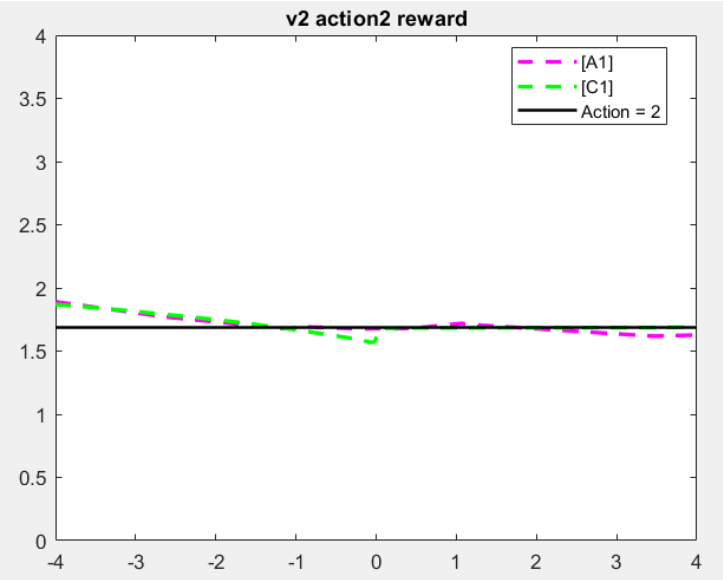
$$\mathrm{Loss\ J} = \frac{1}{\mathrm{n}}\sum_{1}^{n}(\varphi\left(u(x_i,\theta) + Reward(Y) * \psi\big(u(x_i,\theta)\big)\right), \text{όπου } Reward = \min\{2, Y^2\}$$

Το backpropagation και η εκπαίδευση γίνεται όπως έχει αναλυθεί προηγουμένως. Η διαφορά είναι ότι έχουμε m=100 κρυφά επίπεδα. Τα Y data (G(Y) είναι τα rewards).

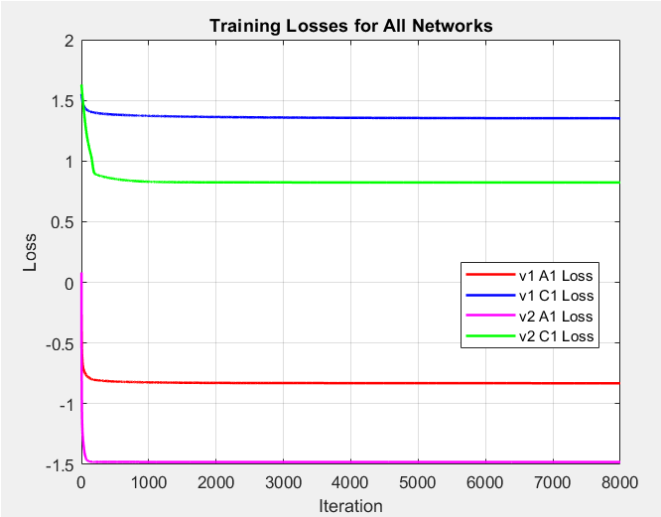
Αποτελέσματα

Για 8000 επαναλήψεις και learning rate 0.001 παίρνουμε τα εξής αποτελέσματα:



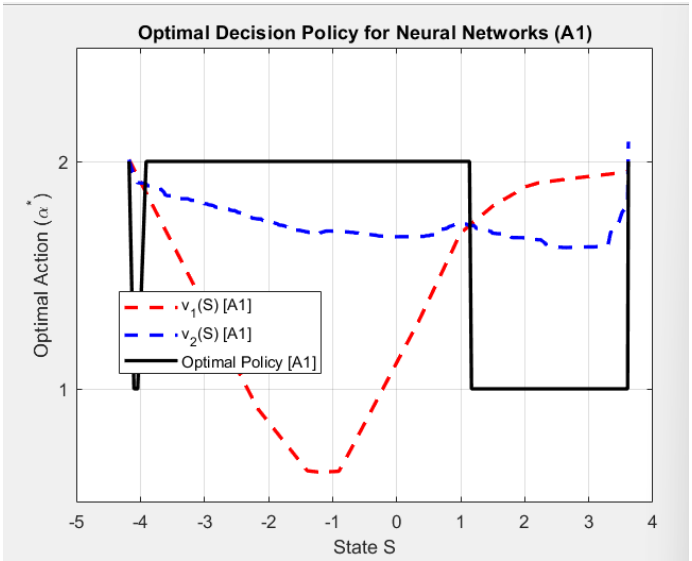


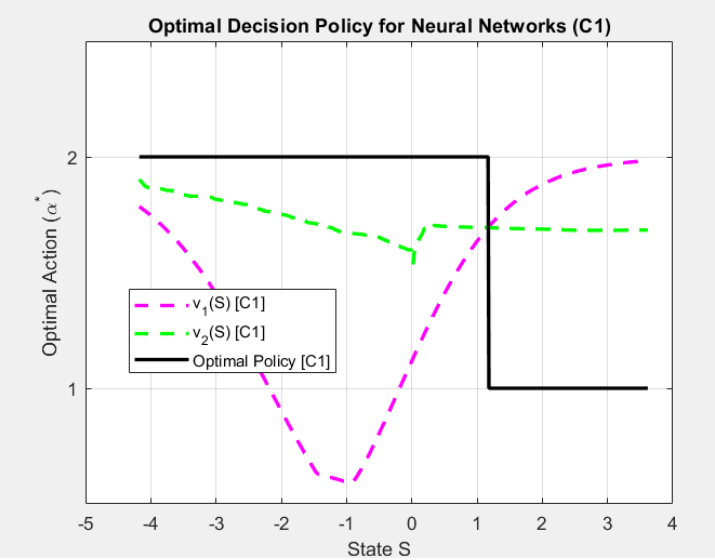
Παρατηρούμε ότι και τα δύο νευρωνικά δίκτυα (A1,C1) κάνουν πολύ καλές προσεγγίσεις με μικρό error και για τα δύο actions. Παρατηρούμε επίσης ότι όλα τα Loss συγκλίνουν:

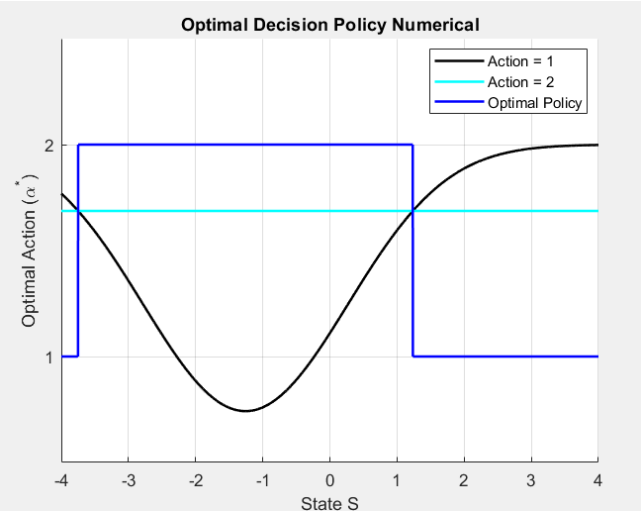


Optimal Decision Policy Comparison

Τα 2 νευρωνικά και η αριθμητική επίλυση καταλήγουν σε σχεδόν πανομοιότυπες Decisions policies. Οι διαφοροποιήσεις συμβαίνουν μόνο στα όρια του διαστήματος (όταν τείνουμε στο -4 και 4, όπου έχουμε ορίσει εμείς ως όρια για τα states S της προσομοίωσης μας, γεγονός λογικό).







Ερώτημα 3

Infinite Future Reward Problem: Δεν κοιτάμε πλέον την μέγιστη επιβράβευση μόνο της επόμενης ενέργειας επομένως έχουμε:

$Q(x) = \max_{a_t, a_{t+1}, \dots} E[R(S_{t+1}) + \gamma R(S_{t+2}) + \gamma^2 R(S_{t+3}) + \cdots | S_t]$, Markovian Property ισχύει άρα κάθε state εξαρτάται μόνο από το προηγούμενο. Επομένως:

$$Q(x) = \max_{a_t} E[R(S_{t+1}) + \gamma Q(S_{t+1}) | S_t = X]$$

Αν γνωρίζουμε Q(X) θα μπορούσαμε να βρούμε την μέση επιβράβευση για κάθε ενέργεια. Καταλήγουμε σε:

$$v_j(x) = E^j_{S_{t+1}}[R(S_{t+1}) + \gamma \max\{v_1(S_{t+1}), \dots, v_K(S_{t+1})\} | S_t = X]$$ (K=2)

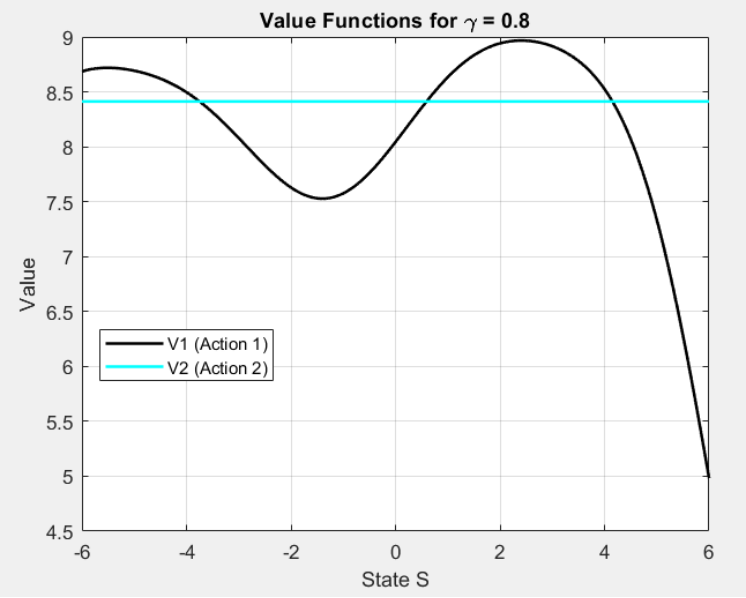
Αριθμητική Επίλυση:

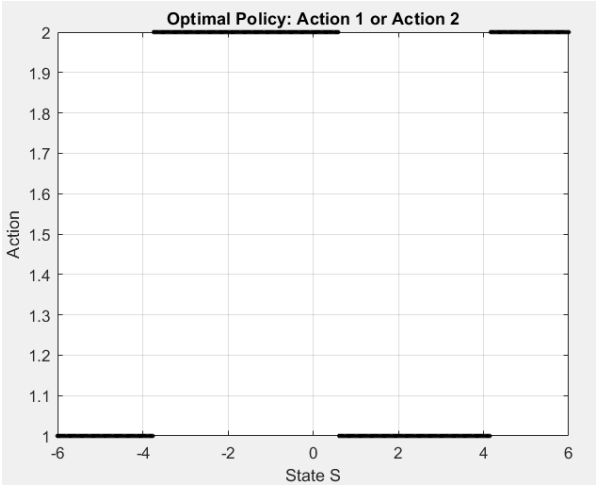
$$V_1 = \begin{bmatrix} v_1(s0) \\ \vdots \\ v_1(sn) \end{bmatrix}, V_2 = \begin{bmatrix} v_2(s0) \\ \vdots \\ v_2(sn) \end{bmatrix}, R = \begin{bmatrix} r(s0) \\ \vdots \\ r(sn) \end{bmatrix},$$

$$\mathrm{Vj} = \mathrm{F}_j\,\mathrm{R} + \gamma \max\{\mathrm{V1}, \dots, \mathrm{VK}\}, \mathrm{j} = 1, \dots, \mathrm{K}$$

Υπολογισμός των μητρών F1,F2, προσοχή όλες οι σειρές να έχουν sum 1. Έχουμε Infinite Horizon Decision Process οπότε το upper limit όταν έχουμε max reward 2 και exponential discounts 0.8 τείνοντας στο άπειρο η τιμή θα είναι 10. Επιβεβαιωνόμαστε και από το γράφημα.

$$V = \sum_{i=0}^{\infty} \gamma^i * R_{max} = R_{max} \sum_{i=0}^{\infty} \gamma^i = 2 * \frac{1}{1-\gamma} \overset{\gamma=0.8}{\Longrightarrow} V = 10$$





Νευρωνικά Δίκτυα

Θα εφαρμόσουμε Stochastic Gradient Descent και για bounded [C1] και για unbounded [A1] νευρωνικά δίκτυα για να προσομοιώσουμε την αριθμητική μέθοδο. Δημιουργούμε όπως στο ερώτημα δύο μια αλληλουχία από States που ανάλογα με το action που οδήγησε σε αυτά ανήκουν στο set1 για a=1 και στο set2 για a=2.

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

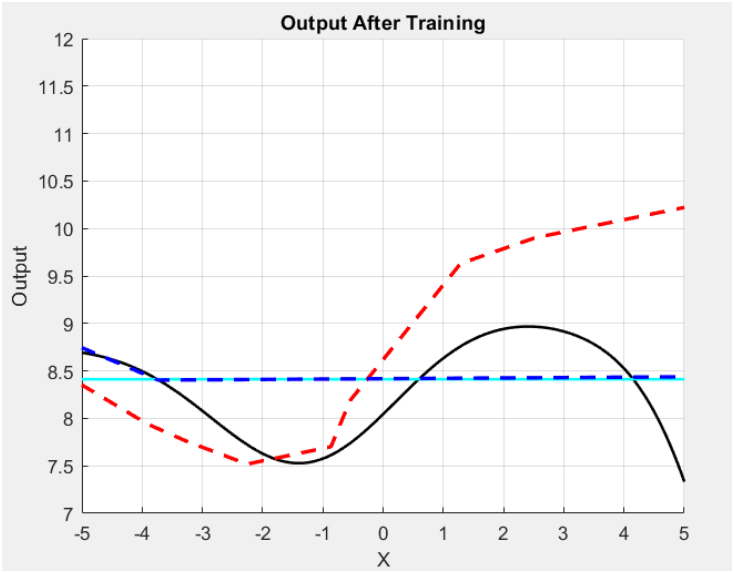
Παρατηρούμε ότι για την εκπαίδευση του το Νευρωνικό Δίκτυο για α=1 χρειάζεται τα Y (S_{t+1}) και των δύο σετ και τα X (S) του set1. Δηλαδή χρειάζονται τα St+1 και των δύο περιπτώσεων για τον υπολογισμό της μελλοντικής επιβράβευσης. Μετά την εκπαίδευση αναμένουμε:

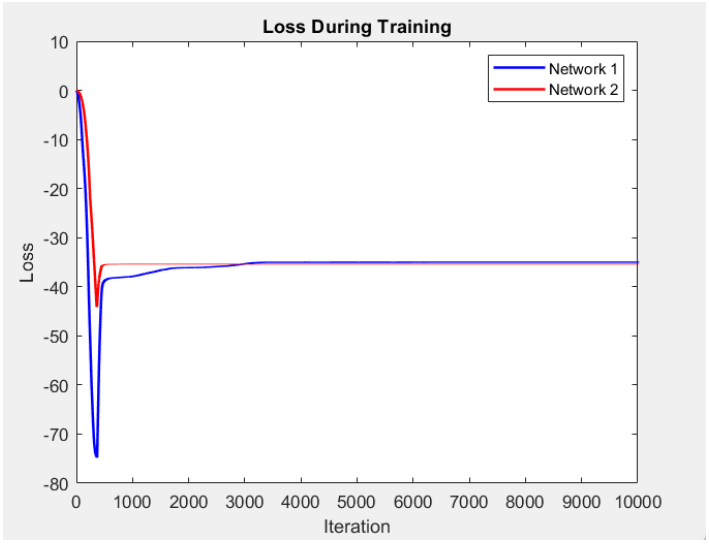
$$\omega(u(X, \theta_o^j)) \approx v_{j=1,2}(X)$$

Όπως και στα προηγούμενα ερωτήματα προφανώς optimal $a = arg\ max_j\{u(X, \theta_o^j)\}$

Stochastic Gradient Descent με A1

Το A1 προσεγγίζει ιδιαίτερα καλά το α=2, ακόμα καλύτερα από ότι στο myopic approach. Η προσέγγιση του α=1 έχει μια απόκλιση, ειδικά όσο πλησιάζουμε στα όρια του διαστήματος των x. Παρόλα αυτά προσεγγίζει ικανοποιητικά την καμπύλη.

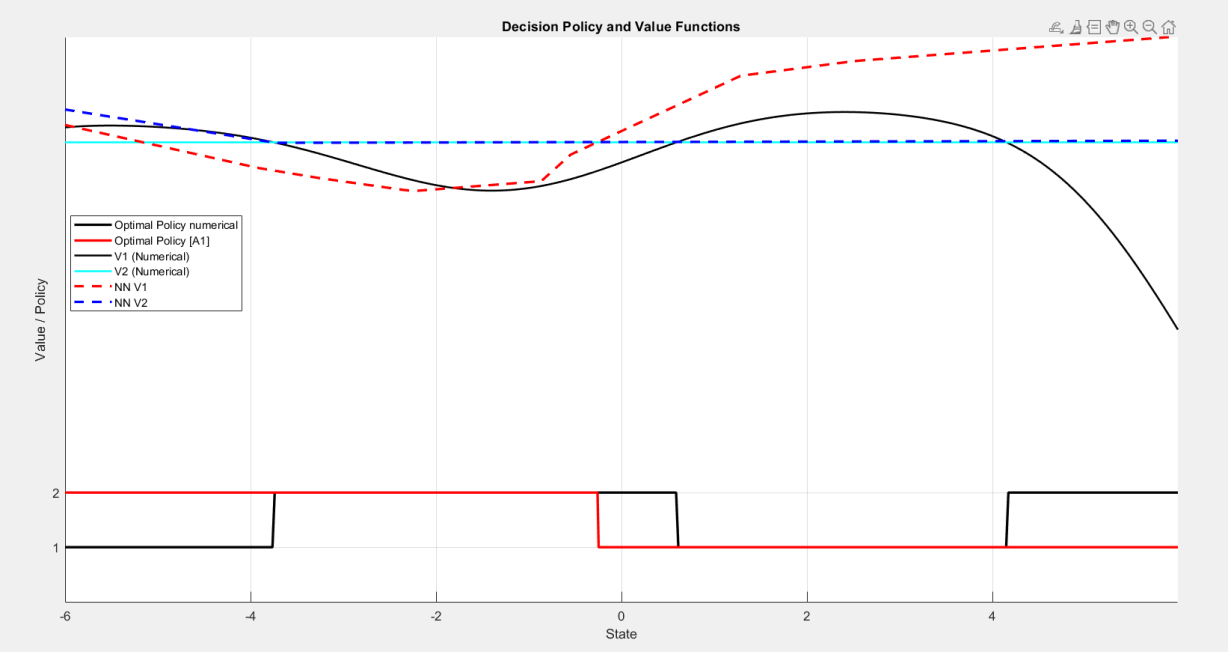




<u>Παρατηρήσεις:</u>

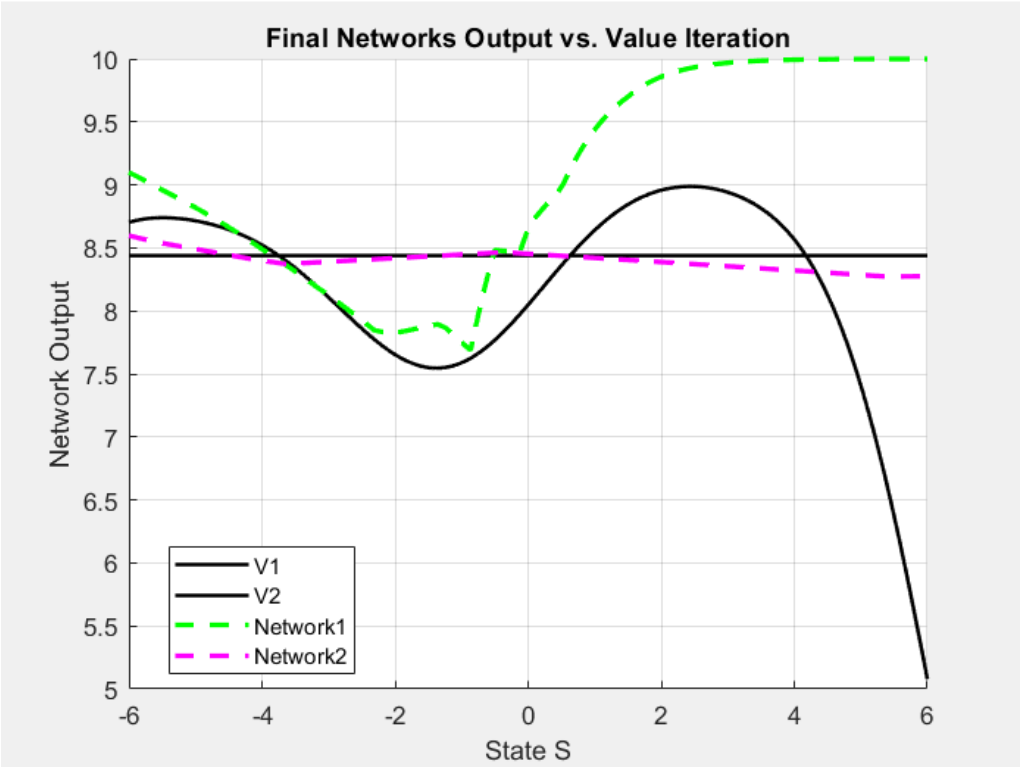
Οι απώλειές συγκλίνουν με τις επιλεγμένες τιμές επαναλήψεων = 10000 και learning rate lr = 0.0002. Όντας όμως **unbounded**, παρατηρούμε ότι η προσέγγιση του A1 για α=1 οδηγεί σε τιμές εκτός του άνω ορίου του 10.

Optimal Policy Decision Comparison: Στα διαστήματα [-5, -3.75], [-0.25,0.6] και [4.17,5] παρατηρούμε ότι το optimal decision του νευρωνικού για α=1 θα διέφερε από την αριθμητική προσέγγιση, δηλαδή στο 29.3% του [-5,5]. Σε αυτά τα σημεία υπάρχουν μεγάλες παράγωγοι (απότομες κλίσεις) οπότε είναι λογικό το ΝΔ να μην προλαβαίνει να «μάθει» την μορφή τους.



Stochastic Gradient Descent με C1

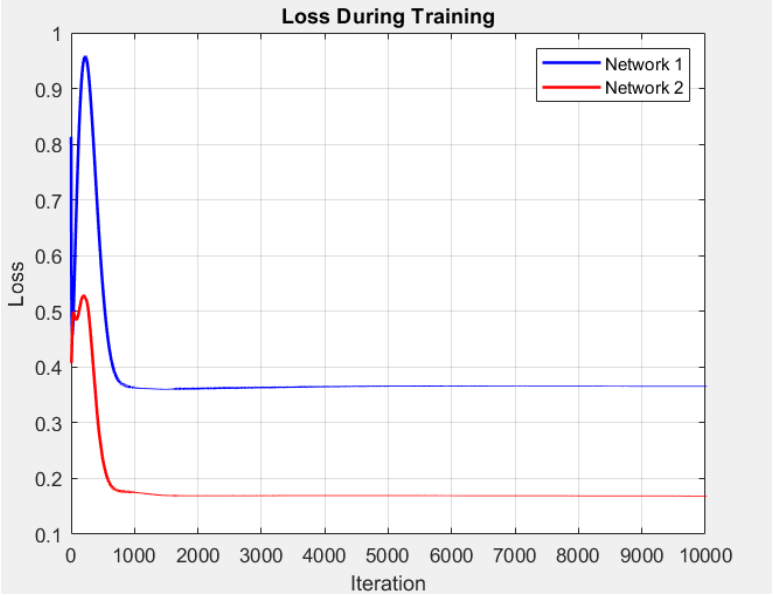
Γνωρίζουμε ότι $v_j(S) \in \left[\frac{c1}{1-\gamma}, \frac{c_2}{1-\gamma}\right]$, επομένως τα όρια a,b του C1 θα είναι α=0, b=10. Εφαρμόζουμε το SGD αλγόριθμο για τις συναρτήσεις C1 και παίρνουμε τα εξής αποτελέσματα.



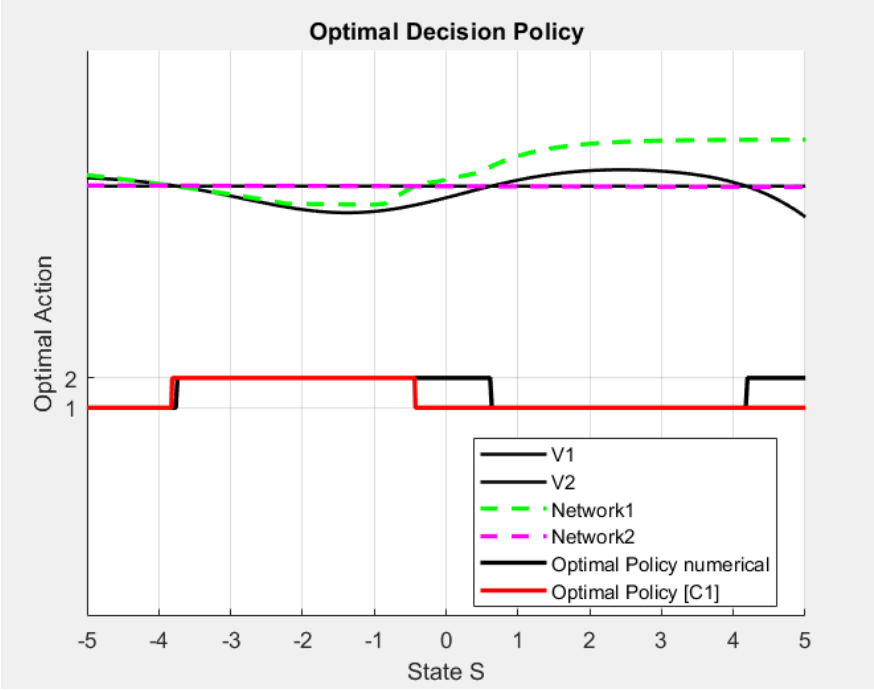
Παρατηρήσεις:

Παρατηρούμε ότι το μαξ όριο του 10 δεν ξεπερνιέται, γεγονός που καθιστά και αυτή την μέθοδο ενδιαφέρουσα και σε αυτό το κομμάτι πιο αξιόπιστη από την Α1. Η προσέγγιση για α=2 είναι πολύ ακριβής. Η προσέγγιση για α=1 έχει ικανοποιητική ακρίβεια και η συνάρτηση του νευρωνικού δικτύου ακολουθεί την μορφή της αριθμητικής προσέγγισης, με εξαίρεση την απότομη πτώση στο [3,6], που δεν ανιχνεύει το C1. Ωστόσο φράσσεται στο 10.

Οι απώλειες και των δύο δικτύων συγκλίνουν για τις τιμές 10000 επαναλήψεων και learning rate= 2*1e-4 που επιλέξαμε.



Optimal Policy Decision Comparison:



Το decision policy διαφέρει μόνο στα διαστήματα [-3.8,-3.7],[-0.5,0.5] και [4.2,5] από τα αριθμητικό, δηλαδή μόνο στο 19% του διαστήματος [-5 ,5] (καλύτερο από το 30% του Α1!).

Σχόλια: Έχει γίνει χρήση σταθεροποιητικού όρου στα gradient updates και clipping για να μην αποσταθεροποιηθεί το αποτέλεσμα από τις μεγάλες παραγώγους. Χωρίς σταθεροποιητικό όρο παρατηρείται περισσότερη ταλάντωση στα losses (αναμενόμενο) έτσι το αποτέλεσμα συγκλίνει πιο αργά και λιγότερο smoothly.

Κώδικας

Ερώτημα 3

Ερώτημα 31α

```
clc;clear;close all;
%r = normrnd(mu,sigma,sz1,...,szN) generates an array of normal random
numbers, where sz1,...,szN indicates the size of each dimension.
n_samples=500;
W = normrnd(0,1,[1,n_samples]);  % Noise ~ N(0,1)
X = linspace(-4, 4, n_samples); %RANDOM X
Y=0.8.*X + W; % Y based on the model Y = 0.8*X + W
% EY[Y|X = X]

% Define grids for X and Y
x_values = linspace(-4, 4, n_samples); % Grid points for X
y_values = linspace(-6, 6, n_samples); % Grid points for Y

% Define conditional CDF H(y|x) (Lecture 11 Equation)
H = @(y, x) normcdf((y - 0.8 * x), 0, 1); % Conditional CDF of Y given X

% Initialize F matrix
F = zeros(n_samples, n_samples);

% Compute F matrix (Trapezoidal Rule Approximation)
for j = 1:n_samples
    % First column
    F(j, 1) = 0.5 * (H(y_values(2), x_values(j)) - H(y_values(1),
x_values(j)));
    % Intermediate columns
    for i = 2:n_samples-1
        F(j, i) = 0.5 * (H(y_values(i+1), x_values(j)) - H(y_values(i-1),
x_values(j)));
    end
    % Last column
    F(j, n_samples) = 0.5 * (H(y_values(n_samples), x_values(j)) -
H(y_values(n_samples-1), x_values(j)));
end

% Normalize F (Row-wise normalization for probabilities)
row_sums = sum(F, 2);
F = F ./ row_sums;

% Define G(Y) functions for expectations
G1_Y = y_values'; % For E[Y|X]
G2_Y = min(1, max(-1, y_values')); % For E[min{1, max{-1, Y}}|X]

% Compute conditional expectations
V1_numeric = F * G1_Y; % Numerical approximation for E[Y|X]
```

```
V2_numeric = F * G2_Y; % Numerical approximation for E[min{1, max{-1,
Y}}|X]
% Plot results
figure;
% Plot E[Y|X]
subplot(1, 2, 1);
plot(x_values, V1_numeric, 'k', 'LineWidth', 2);
title('Part (a): E[Y|X]');
xlabel('X');
ylabel('E[Y|X]');
legend('Numerical', 'Location', 'Best');
% Plot E[min{1, max{-1, Y}}|X]
subplot(1, 2, 2);
plot(x_values, V2_numeric, 'k', 'LineWidth', 2);
title('Part (b): E[min{1, max{-1, Y}}|X]');
xlabel('X');
ylabel('E[min{1, max{-1, Y}}|X]');
legend('Numerical', 'Location', 'Best');
disp('Numerical estimates computed for E[Y|X] and E[min{1, max{-1,
Y}}|X].');
```

Ερώτημα 3.1. E[Y|X=x]

```
clc;clear;close all;
% PART 0: Data Generation
rng(0); % For reproducibility
N = 500; % Number of samples
m = 50; % Number of neurons in the hidden layer
num_iterations = 10000; % Number of gradient descent iterations
%% Alpha
alpha = 0.001;
%alpha = 0.005; % Learning rate

% Generate data: X ~ N(0,1), W ~ N(0,1), Y = 0.8X + W
X_samples = randn(1, N); % 1xN
W_samples = randn(1, N); % 1xN
Y_samples = 0.8 * X_samples + W_samples; % 1xN

% Define the grid for numerical comparison
x_min = floor(min(X_samples));
x_max = ceil(max(X_samples));
x_values = linspace(x_min, x_max, 500); % Grid for X
y_values = linspace(-5, 5, 500); % Grid for Y
H = @(y, x) normcdf((y - 0.8 * x)); % Conditional CDF H(y|x)

% Compute the F-matrix for the numerical solution
F = zeros(500, 500);
for j = 1:500
    F(j, 1) = 0.5 * (H(y_values(2), x_values(j)) - H(y_values(1),
x_values(j)));
    for i = 2:499
        F(j, i) = 0.5 * (H(y_values(i+1), x_values(j)) - H(y_values(i-1),
x_values(j)));
    end
    F(j, 500) = 0.5 * (H(y_values(500), x_values(j)) - H(y_values(499),
x_values(j)));
end
row_sums = sum(F, 2);
F = F ./ row_sums; % Normalize rows of F
```

```
% Numerical solutions
G_Y1 = y_values'; % For E[Y|X]
G_Y2 = min(1, max(-1, y_values')); % For E[G(Y)|X] with G(Y) bounded in [-
1, 1]
V1 = F * G_Y1; % Numerical solution for E[Y|X]
V2 = F * G_Y2; % Numerical solution for E[G(Y)|X]

% Activation function (ReLU)
relu = @(z) max(0, z);
relu_derivative = @(z) (z > 0);

% Initialize weights and biases for [A1], [A2], and [C1]
W1_A1 = randn(m, 1) * 0.1; b1_A1 = zeros(m, 1); W2_A1 = randn(1, m) * 0.1;
b2_A1 = 0;
W1_A2 = randn(m, 1) * 0.1; b1_A2 = zeros(m, 1); W2_A2 = randn(1, m) * 0.1;
b2_A2 = 0;
W1_C1 = randn(m, 1) * 0.1; b1_C1 = zeros(m, 1); W2_C1 = randn(1, m) * 0.1;
b2_C1 = 0;

% Loss values
loss_A1 = zeros(1, num_iterations);
loss_A2 = zeros(1, num_iterations);
loss_C1 = zeros(1, num_iterations);

% PART 1: Train [A1]
phi_A1 = @(z) z.^2 / 2;  % Loss function for [A1]
psi_A1 = @(z) -z;

%alpha=0.002;
for iter = 1:num_iterations
    % Forward pass
    Z1 = W1_A1 * X_samples + b1_A1; % Input to hidden layer
    A1 = relu(Z1); % Hidden layer activations
    Z2 = W2_A1 * A1 + b2_A1; % Output layer
    u_X_A1 = Z2; % Final output

    % Compute loss
    loss_A1(iter) = mean(phi_A1(u_X_A1) + Y_samples .* psi_A1(u_X_A1));

    % Backpropagation
    dZ2 = (u_X_A1 - Y_samples); % Gradient of output layer
    dW2 = (dZ2 * A1')/N ; % Gradient w.r.t W2
    db2 = sum(dZ2)/N ; % Gradient w.r.t b2

    dA1 = W2_A1' * dZ2; % Backprop through W2
    dZ1 = dA1 .* relu_derivative(Z1); % Backprop through ReLU
    dW1 = (dZ1 * X_samples')/N ; % Gradient w.r.t W1
    db1 = sum(dZ1, 2)/N ; % Gradient w.r.t b1

    % Update weights and biases
    W1_A1 = W1_A1 - alpha * dW1;
    b1_A1 = b1_A1 - alpha * db1;
    W2_A1 = W2_A1 - alpha * dW2;
    b2_A1 = b2_A1 - alpha * db2;
end

% PART 2: Train [A2]
phi_A2 = @(z) (exp(0.5*abs(z)) - 1) + (1/3).*(exp(-1.5*abs(z)) - 1);
psi_A2 = @(z) 2 * sign(z) .* (exp(-0.5*abs(z)) - 1);
```

```
for iter = 1:num_iterations
    % Forward pass
    Z1 = W1_A2 * X_samples + b1_A2; % Input to hidden layer
    A1 = relu(Z1); % Hidden layer activations
    Z2 = W2_A2 * A1 + b2_A2; % Output layer
    u_X_A2 = Z2; % Final output

    % Compute loss
    loss_A2(iter) = mean(phi_A2(u_X_A2) + Y_samples .* psi_A2(u_X_A2));

    % Backpropagation
    dZ2 = (u_X_A2 - Y_samples); % Gradient of output layer
    dW2 = (dZ2 * A1')./N ; % Gradient w.r.t W2
    db2 = sum(dZ2) /N; % Gradient w.r.t b2

    dA1 = W2_A2' * dZ2; % Backprop through W2
    dZ1 = dA1 .* relu_derivative(Z1); % Backprop through ReLU
    dW1 = (dZ1 * X_samples')./N ; % Gradient w.r.t W1
    db1 = sum(dZ1, 2)/N ; % Gradient w.r.t b1

    % Update weights and biases
    W1_A2 = W1_A2 - alpha * dW1;
    b1_A2 = b1_A2 - alpha * db1;
    W2_A2 = W2_A2 - alpha * dW2;
    b2_A2 = b2_A2 - alpha * db2;
end
omega= @(z) z;
omega2= @(z) sinh(z);
% PART 3: Compare and Plot Results
figure;
subplot(2, 1, 1);
plot(x_values, V1, 'k-', 'LineWidth', 2, 'DisplayName', 'Numeric E[Y|X]');
hold on;
plot(x_values, omega(W2_A1 * relu(W1_A1 * x_values + b1_A1)), 'r--',
'LineWidth', 2, 'DisplayName', '[A1]');
plot(x_values, omega2(W2_A2 * relu(W1_A2 * x_values + b1_A2)), 'b-.',
'LineWidth', 2, 'DisplayName', '[A2]');
legend('show');
xlabel('X');
ylabel('E[Y|X]');
title('Part (α): E[Y|X]');

subplot(2, 1, 2);
plot(1:num_iterations, loss_A1, 'r-', 'LineWidth', 1.5, 'DisplayName',
'Loss [A1]');
hold on;
plot(1:num_iterations, loss_A2, 'b-', 'LineWidth', 1.5, 'DisplayName',
'Loss [A2]');
xlabel('Iterations');
ylabel('Loss');
title('Loss During Training');
legend('show');
grid on;
```

Ερώτημα 3.1 E[min{1,max{−1,Y}}|X=x]

```
clc;clear;close all;

% PART 0: Data Generation
```

```
rng(0); % For reproducibility

N = 500; % Number of samples
m = 50; % Number of neurons in the hidden layer
num_iterations = 5000; % Number of gradient descent iterations
alpha = 0.01; % Learning rate
alphac=0.01; %different for c1
% Generate data: X ~ N(0,1), W ~ N(0,1), Y = 0.8X + W
X_samples = randn(1, N); % 1xN
W_samples = randn(1, N); % 1xN
Y_samples = 0.8 * X_samples + W_samples; % 1xN

% Define the grid for numerical comparison
x_min = floor(min(X_samples));
x_max = ceil(max(X_samples));
x_values = linspace(x_min, x_max, 500); % Grid for X
y_values = linspace(-5, 5, 500); % Grid for Y
H = @(y, x) normcdf((y - 0.8 * x)); % Conditional CDF H(y|x)

% Compute the F-matrix for the numerical solution
F = zeros(500, 500);
for j = 1:500
    F(j, 1) = 0.5 * (H(y_values(2), x_values(j)) - H(y_values(1),
x_values(j)));
    for i = 2:499
        F(j, i) = 0.5 * (H(y_values(i+1), x_values(j)) - H(y_values(i-1),
x_values(j)));
    end
    F(j, 500) = 0.5 * (H(y_values(500), x_values(j)) - H(y_values(499),
x_values(j)));
end
row_sums = sum(F, 2);
F = F ./ row_sums; % Normalize rows of F

% Numerical solution for bounded expectation
G_Y = min(1, max(-1, y_values')); % G(Y) = min(1, max(-1, Y))
V = F * G_Y; % Numerical solution for E[G(Y)|X]

% Activation function (ReLU)
relu = @(z) max(0, z);
relu_derivative = @(z) (z > 0);

% Initialize weights and biases for [A1] and [C1]
W1_A1 = randn(m, 1) * 0.1; b1_A1 = zeros(m, 1); W2_A1 = randn(1, m) * 0.1;
b2_A1 = 0;
W1_C1 = randn(m, 1) * 0.1; b1_C1 = zeros(m, 1); W2_C1 = randn(1, m) * 0.1;
b2_C1 = 0;

% Loss values
loss_A1 = zeros(1, num_iterations);
loss_C1 = zeros(1, num_iterations);
g=@(y)min(1, max(-1, y));
omega = @(z) z;
%% PART 1: Train [A1]
phi_A1 = @(z) z.^2 / 2;  % Loss function for [A1]
psi_A1 = @(z) -z;

for iter = 1:num_iterations
    % Forward pass
```

```
    Z1 = W1_A1 * X_samples + b1_A1; % Input to hidden layer
    A1 = relu(Z1); % Hidden layer activations
    Z2 = W2_A1 * A1 + b2_A1; % Output layer
    u_X_A1 = Z2; % Final output

    % Compute loss
    loss_A1(iter) = mean(phi_A1(u_X_A1) + g(Y_samples) .* psi_A1(u_X_A1));

    % Backpropagation
    dZ2 = (u_X_A1 - Y_samples); % Gradient of output layer
    dW2 = (dZ2 * A1') / N; % Gradient w.r.t W2
    db2 = sum(dZ2) / N; % Gradient w.r.t b2

    dA1 = W2_A1' * dZ2; % Backprop through W2
    dZ1 = dA1 .* relu_derivative(Z1); % Backprop through ReLU
    dW1 = (dZ1 * X_samples') / N; % Gradient w.r.t W1
    db1 = sum(dZ1, 2) / N; % Gradient w.r.t b1

    % Update weights and biases
    W1_A1 = W1_A1 - alpha * dW1;
    b1_A1 = b1_A1 - alpha * db1;
    W2_A1 = W2_A1 - alpha * dW2;
    b2_A1 = b2_A1 - alpha * db2;
end

%% PART 2: Train [C1]
phi_C1 = @(z)  2./(1+exp(z))+log(1+exp(z)); %@(z) (z - 1).^2 .* (z > 1) +
(z + 1).^2 .* (z < -1); % Penalize outside [-1, 1]
psi_C1 =  @(z) -log(1+exp(z));  %@(z) 2 * max(-1, min(1, z)); % Ensure
bounded range
omega2 = @(z)(-1./ (1 + exp(z)))+ (exp(z) ./ (1 + exp(z)));
W1_C1 = randn(m, 1) * 0.001; % Smaller initial weights
b1_C1 = zeros(m, 1);
W2_C1 = randn(1, m) * 0.001;
b2_C1 = 0;

for iter = 1:num_iterations
    % Forward pass
    Z1 = W1_C1 * X_samples + b1_C1; % Input to hidden layer
    A1 = relu(Z1); % Hidden layer activations
    Z2 = W2_C1 * A1 + b2_C1; % Output layer
    %u_X_C1 = Z2; % Final output
    %[a,b]=[-1,1]
    u_X_C1=-(1./(1+exp(-Z2'))).*(Y_samples-1./(1+exp(-Z2'))-(-
1)./(1+exp(Z2')));

    % Compute loss
    %loss_C1(iter) = mean(phi_C1(u_X_C1) + g(Y_samples) .* psi_C1(u_X_C1));
    %loss_C1(iter) = mean(phi_C1(u_X_C1(:)) + g(Y_samples(:)) .*
psi_C1(u_X_C1(:)));
    loss_C1(iter)=mean((2)./(1+exp(Z2'))+1*log(1+exp(Z2'))-
Y_samples'.*log(1+exp(Z2')));
    % Backpropagation
    %dZ2 = (u_X_C1 - Y_samples); % Gradient of output layer
    dZ2 = -(2 * exp(Z2) ./ (1 + exp(Z2)).^2) + (exp(Z2) ./ (1 + exp(Z2))) -
Y_samples .* (exp(Z2) ./ (1 + exp(Z2)));

    dW2 = (dZ2 * A1') / N; % Gradient w.r.t W2
    db2 = sum(dZ2) / N; % Gradient w.r.t b2
```

```
    dA1 = W2_C1' * dZ2; % Backprop through W2
    dZ1 = dA1 .* relu_derivative(Z1); % Backprop through ReLU
    dW1 = (dZ1 * X_samples') / N; % Gradient w.r.t W1
    db1 = sum(dZ1, 2) / N; % Gradient w.r.t b1

    % Update weights and biases
    W1_C1 = W1_C1 - alphac * dW1;
    b1_C1 = b1_C1 - alphac * db1;
    W2_C1 = W2_C1 - alphac * dW2;
    b2_C1 = b2_C1 - alphac * db2;
end

% PART 3: Compare and Plot Results
figure;
subplot(2, 1, 1);
plot(x_values, V, 'k-', 'LineWidth', 2, 'DisplayName', 'Numeric
E[G(Y)|X]');
hold on;
plot(x_values, omega(W2_A1 * relu(W1_A1 * x_values + b1_A1)), 'r--',
'LineWidth', 2, 'DisplayName', '[A1]');
plot(x_values, omega2(W2_C1 * relu(W1_C1 * x_values + b1_C1)), 'b-.',
'LineWidth', 2, 'DisplayName', '[C1]');
legend('show');
xlabel('X');
ylabel('E[G(Y)|X]');
title('E[G(Y)|X], G(Y) = min(1, max(-1, Y))');

subplot(2, 1, 2);
plot(1:num_iterations, loss_A1, 'r-', 'LineWidth', 1.5, 'DisplayName',
'Loss [A1]');
hold on;
plot(1:num_iterations, loss_C1, 'b-', 'LineWidth', 1.5, 'DisplayName',
'Loss [C1]');
xlabel('Iterations');
ylabel('Loss');
title('Loss During Training');
legend('show');
grid on;
```

Ερώτημα 2

```
clc;clear;close all;
clc;clear;close all;
rng(0);
%% SAMPLES for understanding
% N = 1000;
% % Generate 1000 samples, each either 1 or 2 with 50/50 probability
% action = randi([1, 2], 1, N);
% % 2 sets
% set1 =[];
% set2=[];
% % Initial state
% state = randn;  % Random initial state from a standard normal
distribution
% % DECISIONS
% for i = 1:N
%     W = randn; %noise
%     % Decision logic based on the action
```

```
%     if action(i) == 1
%         next_state = 0.8 * state + 1 + W;
%         set1=[set1;state,next_state];
%     elseif action(i) == 2
%         next_state = -2 + W;
%         set2=[set2;state,next_state];
%     else
%         error('Invalid input: a must be either 1 or 2.');
%     end
%     state = next_state;
% end

%% Helpers
reward = @(y) min(2, y.^2);
% normcdf(x,mu,sigma) returns the cdf of the normal distribution with mean
mu and standard deviation sigma, evaluated at the values in x.
H1 = @(st1, st) normcdf(st1, 0.8 * st + 1, 1); % H(Y | X) = Φ((Y - (0.8S_t
+ 1)) / 1)
H2 = @(st1, st) normcdf(st1, -2, 1); % H(Y | X) = Φ((Y + 2) / 1)

%% NUMERICAL U = F * G
% Define state space
n = 3000;  % Number of discretized states
X = linspace(-4, 4, n);  % State space for S_t (X GRID)
Y = linspace(-8.2, 8.2, n);  % State space for S_{t+1} (Y GRID)

% Define reward vector R(S) = min(2, S^2)
R = reward(Y)';

% Transition matrices F1 and F2
F1 = zeros(n, n);  % For action = 1
F2 = zeros(n, n);  % For action = 2

% Compute F1 for action = 1
for j = 1:n
    F1(j, 1) = 0.5 * (H1(Y(2), X(j)) - H1(Y(1), X(j)));
    F1(j, 3:(n-1)) = 0.5 * (H1(Y(3:(n-1)), X(j)) - H1(Y(1:(n-3)), X(j)));
    F1(j, n) = 0.5 * (H1(Y(n), X(j)) - H1(Y(n-1), X(j)));
end

% Compute F2 for action = 2
for j = 1:n
    F2(j, 1) = 0.5 * (H2(Y(2), X(j)) - H2(Y(1), X(j)));
    F2(j, 3:(n-1)) = 0.5 * (H2(Y(3:(n-1)), X(j)) - H2(Y(1:(n-3)), X(j)));
    F2(j, n) = 0.5 * (H2(Y(n), X(j)) - H2(Y(n-1), X(j)));
end

% Compute rewards for each action
V1 = F1 * R;  % Reward for action = 1
V2 = F2 * R;  % Reward for action = 2
% Compute Optimal Policy
optimal_policy = zeros(1, length(X)); % Initialize optimal policy array
for i = 1:length(X)
    if V1(i) >= V2(i)
        optimal_policy(i) = 1; % Action = 1
    else
        optimal_policy(i) = 2; % Action = 2
    end
end
```

```
% Plot the results
figure;
hold on; grid on;
plot(X, V1, 'k-', 'LineWidth', 1.5, 'DisplayName', 'Action = 1');
plot(X, V2, 'c-', 'LineWidth', 1.5, 'DisplayName', 'Action = 2');
xlabel('State S'); ylabel('Reward E[R(S_{t+1}) | S_t]');
legend('Location', 'best');
title('Reward Comparison for Action = 1 and Action = 2');

% Plot Optimal Policy
figure;
hold on; grid on;
plot(X, V1, 'k-', 'LineWidth', 1.5, 'DisplayName', 'Action = 1');
plot(X, V2, 'c-', 'LineWidth', 1.5, 'DisplayName', 'Action = 2');
plot(X, optimal_policy, 'b-', 'LineWidth', 1.5, 'DisplayName', 'Optimal
Policy');
xlabel('State S');
%legend('Location', 'best');
%title('Reward Comparison for Action = 1 and Action = 2');

ylim([0.5 2.5]);
yticks([1 2]);
ylabel('Optimal Action (\alpha^*)');
title('Optimal Decision Policy');
legend('Location', 'best');
grid on;

%% DATA DRIVEN
%% Data gen
N = 1000;
% Generate 1000 samples, each either 1 or 2 with 50/50 probability
action = randi([1, 2], 1, N);
% 2 sets
set1 =[];
set2=[];
% Initial state
state = randn;  % Random initial state from a standard normal distribution
% DECISIONS
for i = 1:N
    W = randn; %noise
    % Decision logic based on the action
    if action(i) == 1
        next_state = 0.8 * state + 1 + W;
        set1=[set1;state,next_state];
    elseif action(i) == 2
        next_state = -2 + W;
        set2=[set2;state,next_state];
    else
        error('Invalid input: a must be either 1 or 2.');
    end
    state = next_state;
end

X1_data= set1(:,1)';
Y1_data= set1(:,2)';
R1_data= reward(Y1_data);

X2_data= set2(:,1)';
```

```
Y2_data= set2(:,2)';
R2_data= reward(Y2_data);

%% Neural Network
%parameters
m=100;
num_iter= 8000;
alpha= 0.001;%learning rate
%[A1]
phi_A1 = @(z) z.^2 / 2;  % Loss function for [A1]
psi_A1 = @(z) -z;
omega1= @(z) z;
%[C1]
phi_C1 = @(z)  2./(1+exp(z))+log(1+exp(z)); %@(z) (z - 1).^2 .* (z > 1) +
(z + 1).^2 .* (z < -1); % Penalize outside [-1, 1]
psi_C1 =  @(z) -log(1+exp(z));  %@(z) 2 * max(-1, min(1, z)); % Ensure
bounded range
omega_C1 = @(z)(-1./ (1 + exp(z)))+ (exp(z) ./ (1 + exp(z)));
% Activation function (ReLU)
relu = @(z) max(0, z);
relu_derivative = @(z) (z > 0);

% Initialize weights and biases for [A1]v1,v2,and [C1]v2,v2
cost_v1A1= zeros(1,num_iter);
cost_v1C1= zeros(1,num_iter);
cost_v2A1= zeros(1,num_iter);
cost_v2C1= zeros(1,num_iter);

% v1(A1)
w1_v1A1= 0.01*randn(m,1);
b1_v1A1= zeros(m,1);
w2_v1A1= randn(1,m);
b2_v1A1= 0;

% v1(C1)
w1_v1C1= randn(m,1);
b1_v1C1= zeros(m,1);
w2_v1C1= randn(1,m);
b2_v1C1= 0;

% v2(A1)
w1_v2A1= 0.01*randn(m,1);
b1_v2A1= zeros(m,1);
w2_v2A1= randn(1,m);
b2_v2A1= 0;

% v2(C1)
w1_v2C1= randn(m,1);
b1_v2C1= zeros(m,1);
w2_v2C1= randn(1,m);
b2_v2C1= 0;

% Loss values
loss_v1A1 = zeros(1, num_iter);
loss_v2A1 = zeros(1, num_iter);
loss_v1C1 = zeros(1, num_iter);
loss_v2C1 = zeros(1, num_iter);
```

```
[~, ~, ~, ~, loss_v1A1,outputA1V1] = A1_training(X1_data, R1_data, w1_v1A1,
b1_v1A1, w2_v1A1, b2_v1A1, alpha, num_iter, relu,loss_v1A1);
[X1_sort, idx1] = sort(X1_data);
yA1_1=outputA1V1(idx1);
[~, ~, ~, ~, loss_v1C1,outputC1V1]= C1_training(X1_data, R1_data, w1_v1A1,
b1_v1A1, w2_v1A1, b2_v1A1, alpha, num_iter, relu,loss_v1C1);
yC1_1=outputC1V1(idx1);

[~, ~, ~, ~, loss_v2A1,outputA1V2] = A1_training(X2_data, R2_data, w1_v2A1,
b1_v2A1, w2_v2A1, b2_v2A1, alpha, num_iter, relu,loss_v2A1);
[X2_sort, idx2] = sort(X2_data);
yA1_2=outputA1V2(idx2);
[w1, b1, w2, b2, loss_v2C1,outputC1V2] = C1_training(X2_data, R2_data,
w1_v2C1, b1_v2C1, w2_v2C1, b2_v2C1, alpha, num_iter, relu,loss_v2C1);
yC1_2=outputC1V2(idx2);

figure;
plot(X1_sort, yA1_1, 'r--','LineWidth',2,'DisplayName','[A1]');
hold on
plot(X1_sort, yC1_1, 'b--','LineWidth',2,'DisplayName','[C1]');
hold on
plot(X, V1, 'k-', 'LineWidth', 1.5, 'DisplayName', 'Action = 1');
legend('Location', 'Best');
xlim([-4,4]);
title("v1 action1 reward");
figure;
plot(X2_sort, yA1_2, 'm--','LineWidth',2,'DisplayName','[A1]');
hold on
plot(X2_sort, yC1_2, 'g--','LineWidth',2,'DisplayName','[C1]');
hold on
plot(X, V2, 'k-', 'LineWidth', 1.5, 'DisplayName', 'Action = 2');
legend('Location', 'Best');
xlim([-4,4]);
ylim([0 4]);
title("v2 action2 reward");
% Plot all losses
figure;
plot(1:num_iter, loss_v1A1, 'r-', 'LineWidth', 1.5, 'DisplayName', 'v1 A1
Loss');
hold on;
plot(1:num_iter, loss_v1C1, 'b-', 'LineWidth', 1.5, 'DisplayName', 'v1 C1
Loss');
plot(1:num_iter, loss_v2A1, 'm-', 'LineWidth', 1.5, 'DisplayName', 'v2 A1
Loss');
plot(1:num_iter, loss_v2C1, 'g-', 'LineWidth', 1.5, 'DisplayName', 'v2 C1
Loss');
xlabel('Iteration');
ylabel('Loss');
legend('Location', 'Best');
title('Training Losses for All Networks');
grid on;

% Compute Optimal Policy
% Align lengths for A1
len_a1 = min(length(X1_sort), length(yA1_2)); % Use the shortest length
optimal_policy_nn = zeros(1, len_a1); % Initialize optimal policy array

for i = 1:len_a1
    if yA1_1(i) >= yA1_2(i)
```

```
        optimal_policy_nn(i) = 1; % Action = 1
    else
        optimal_policy_nn(i) = 2; % Action = 2
    end
end

% Plot Optimal Decision Policy for Neural Networks (A1)
figure;
plot(X1_sort(1:len_a1), yA1_1(1:len_a1), 'r--', 'LineWidth', 2,
'DisplayName', 'v_1(S) [A1]');
hold on;
plot(X1_sort(1:len_a1), yA1_2(1:len_a1), 'b--', 'LineWidth', 2,
'DisplayName', 'v_2(S) [A1]');
hold on;
plot(X1_sort(1:len_a1), optimal_policy_nn, 'k-', 'LineWidth', 2,
'DisplayName', 'Optimal Policy [A1]');
xlabel('State S');
ylabel('Optimal Action (\alpha^*)');
ylim([0.5 2.5]);
yticks([1 2]);
title('Optimal Decision Policy for Neural Networks (A1)');
legend('Location', 'best');
grid on;

% Align lengths for C1
len_c1 = min(length(X1_sort),  length(yC1_2)); % Use the shortest length
optimal_policy_nn_c1 = zeros(1, len_c1); % Initialize optimal policy array

for i = 1:len_c1
    if yC1_1(i) >= yC1_2(i)
        optimal_policy_nn_c1(i) = 1; % Action = 1
    else
        optimal_policy_nn_c1(i) = 2; % Action = 2
    end
end

% Plot Optimal Decision Policy for Neural Networks (C1)
figure;
plot(X1_sort(1:len_c1), yC1_1(1:len_c1), 'm--', 'LineWidth', 2,
'DisplayName', 'v_1(S) [C1]');
hold on;
plot(X1_sort(1:len_c1), yC1_2(1:len_c1), 'g--', 'LineWidth', 2,
'DisplayName', 'v_2(S) [C1]');
hold on;
plot(X1_sort(1:len_c1), optimal_policy_nn_c1, 'k-', 'LineWidth', 2,
'DisplayName', 'Optimal Policy [C1]');
xlabel('State S');
ylabel('Optimal Action (\alpha^*)');
ylim([0.5 2.5]);
yticks([1 2]);
title('Optimal Decision Policy for Neural Networks (C1)');
legend('Location', 'best');
grid on;


function [w1, b1, w2, b2,loss,output] = A1_training(X_data, R_data, w1, b1,
w2, b2, lr, num_iter, ReLU,loss)
    phi_A1 = @(z) z.^2 / 2;  % Loss function for [A1]
    psi_A1 = @(z) -z;
```

```
    omega1= @(z) z;
    % Number of samples
    N = length(X_data);
    m = size(w1, 1);  % Number of hidden units

    % Initialize cost storage
    %cost = zeros(1, num_iter);

    for iter = 1:num_iter
        % Forward pass
        Z1 = w1 * X_data + b1;  % (m x #samples)
        A1 = ReLU(Z1);          % ReLU activation
        Z2 = w2 * A1 + b2;      % (1 x #samples)
        A2 = Z2;                % Identity activation
        output=omega1(A2);
        N= length(X_data);
        % Compute cost
        %cost(iter) = mean(A2.^2 / 2 - R_data .* A2);
        loss(iter) = mean(phi_A1(A2) + R_data .* psi_A1(A2));
        % Backpropagation
        dZ2 = A2 - R_data;              % (1 x N)
        dW2 = (dZ2 * A1') / N;          % (1 x m)
        db2 = sum(dZ2, 2) / N;          % Scalar

        dA1 = w2' * dZ2;                % (m x N)
        dZ1 = dA1 .* (Z1 > 0);          % ReLU derivative
        dW1 = (dZ1 * X_data') / N;      % (m x 1)
        db1 = sum(dZ1, 2) / N;          % (m x 1)

        % Gradient descent update
        w2 = w2 - lr * dW2;
        b2 = b2 - lr * db2;
        w1 = w1 - lr * dW1;
        b1 = b1 - lr * db1;
    end
end

function [w1, b1, w2, b2,loss,output] = C1_training(X_data, R_data, w1, b1,
w2, b2, lr, num_iter, ReLU,loss)
    phi_C1 = @(z)  2./(1+exp(z))+2.*log(1+exp(z)); %@(z) (z - 1).^2 .* (z >
1) + (z + 1).^2 .* (z < -1); % Penalize outside [-1, 1]
    psi_C1 =  @(z) -log(1+exp(z));  %@(z) 2 * max(-1, min(1, z)); % Ensure
bounded range
    omega_C1 = @(z) 2.* exp(z)./(1+ exp(z));
    % Number of samples
    N = length(X_data);
    m = size(w1, 1);  % Number of hidden units

    % Initialize cost storage
    %cost = zeros(1, num_iter);

    for iter = 1:num_iter
        % Forward pass
        Z1 = w1 * X_data + b1;  % (m x #samples)
        A1 = ReLU(Z1);          % ReLU activation
        Z2 = w2 * A1 + b2;      % (1 x #samples)
        A2 = Z2;                % Identity activation
        output=omega_C1(A2);
        % Compute cost
```

```
        %cost(iter) = mean(A2.^2 / 2 - R_data .* A2);
        loss(iter) = mean(phi_C1(A2) + (R_data) .* psi_C1(A2));
        % Backpropagation
        dZ2 = -2.* exp(A2)./(1+exp(A2)).^2 + ...
                  (2 - R_data).* exp(A2)./(1+exp(A2));              % (1 x
N)
        % dZ2 =-2.* exp(A2)./(1+exp(A2)).^2 + ...
        %           (2 - R_data).* exp(A2)./(1+exp(A2));
        dW2 = (dZ2 * A1') / N;           % (1 x m)
        db2 = sum(dZ2, 2) / N;           % Scalar

        dA1 = w2' * dZ2;                 % (m x N)
        dZ1 = dA1 .* (Z1 > 0);           % ReLU derivative
        dW1 = (dZ1 * X_data') / N;       % (m x 1)
        db1 = sum(dZ1, 2) / N;           % (m x 1)

        % Gradient descent update
        w2 = w2 - lr * dW2;
        b2 = b2 - lr * db2;
        w1 = w1 - lr * dW1;
        b1 = b1 - lr * db1;
    end
end
```

Ερώτημα 3

Αριθμητική Λύση

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Numeric solution (Infinite Horizon)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 clc; close all;

%% (0) Helper definitions
reward = @(y) min(2, y.^2);

% Transition CDFs for actions
% H1(s_{t+1}, s_t) = Phi( s_{t+1} - [0.8*s_t + 1] )
% H2(s_{t+1}, s_t) = Phi( s_{t+1} - (-2) )
H1 = @(st1, st) normcdf(st1, 0.8*st + 1, 1);
H2 = @(st1, st) normcdf(st1, -2,         1);

gamma  = 0.8;    % discount factor
tol    = 1e-6;   % convergence tolerance
maxIter= 1000;    % max number of iterations

%% (1) Discretize State Spaces
n  = 500;
X  = linspace(-6,  6,   n);   % possible "current" states S_t
%Y  = linspace(-10,12, n);   % possible "next" states S_{t+1}
Y=X;
Rvec = min(2, Y'.^2);
%Rvec = reward(Y)';

%% (2) Build Transition Matrices F1, F2  (each n x n)
F1 = zeros(n, n);
F2 = zeros(n, n);

for j = 1:n
```

```
    F1(j,1) = 0.5 * ( H1(Y(2), X(j)) - H1(Y(1), X(j)) );
    F2(j,1) = 0.5 * ( H2(Y(2), X(j)) - H2(Y(1), X(j)) );

    for k = 2:(n-1)
        F1(j,k) = 0.5 * ( H1(Y(k+1), X(j)) - H1(Y(k), X(j)) );
        F2(j,k) = 0.5 * ( H2(Y(k+1), X(j)) - H2(Y(k), X(j)) );
    end

    F1(j,n) = 0.5 * ( H1(Y(n),   X(j)) - H1(Y(n-1), X(j)) );
    F2(j,n) = 0.5 * ( H2(Y(n),   X(j)) - H2(Y(n-1), X(j)) );

end

%% (3) Value Iteration
V1 = zeros(n,1);   % n x 1
V2 = zeros(n,1);   % n x 1

for iter = 1:maxIter
    oldV1 = V1;
    oldV2 = V2;

    % bestNext is also (n x 1)
    bestNext = max(oldV1, oldV2);

    % Rvec is (n x 1), bestNext is (n x 1), so Rvec + gamma*bestNext is (n
x 1).
    % F1, F2 are (n x n). => (n x n) * (n x 1) => (n x 1).
    V1 = F1 * ( Rvec + gamma * bestNext );
    V2 = F2 * ( Rvec + gamma * bestNext );
    targetRow= Rvec + gamma.* bestNext;
    new1(iter)= sum( F1(iter,:)' .* targetRow );
    new2(iter)= sum( F2(iter,:)' .* targetRow );
    % Check convergence
    diff1 = max(abs(V1 - oldV1));
    diff2 = max(abs(V2 - oldV2));
    if max(diff1, diff2) < tol
        fprintf('Converged at iteration %d\n', iter);
        break;
    end
end

%% (4) Plot results
figure('Name','Infinite Horizon V1 vs V2');
plot(X, V1, 'k-', 'LineWidth',1.5); hold on; grid on;
plot(X, V2, 'r-', 'LineWidth',1.5);
legend('V1','V2','Location','Best');
xlabel('State S'); ylabel('Value');
xlim([-6,6]);
title('Numerical Value Functions for \gamma=0.8');

% Optional: Optimal policy
policy = ones(n,1);
policy(V2 > V1) = 2;
figure('Name','Optimal Policy');
plot(X, policy, 'b.', 'MarkerSize',8); grid on;
xlabel('State'); ylabel('Action');
title('Policy = 1 or 2');
```

Αριθμητική Λύση και Data Driven A1

```
clc; close all;
rng(0);
%% Helper Definitions
reward = @(y) min(2, y.^2);

% Transition CDFs for actions
H1 = @(st1, st) normcdf(st1, 0.8*st + 1, 1);
H2 = @(st1, st) normcdf(st1, -2, 1);

gamma = 0.8;  % Discount factor
tol = 1e-6;   % Convergence tolerance
maxIter = 1000;  % Max number of iterations

%% (1) Discretize State Space
num_states = 500;
state_range = linspace(-6, 6, num_states);  % Discrete states S_t
R = reward(state_range)';  % Reward for each state (column vector)

%% (2) Build Transition Matrices F1, F2
F1 = zeros(num_states, num_states);
F2 = zeros(num_states, num_states);
mw = 0; % Mean of noise
sigma_w = 1; % Standard deviation of noise

for j = 1:num_states
    S_t = state_range(j);
    for i = 1:num_states
        S_t1 = state_range(i);
        if i == 1
            F1(j, i) = 0.5 * (normcdf(S_t1, 0.8 * S_t + 1 + 0, 1) -
normcdf(S_t1 - (20 / num_states), 0.8 * S_t + 1 + 0, 1));
            F2(j, i) = 0.5 * (normcdf(S_t1,  -2 + 0, 1) - normcdf(S_t1 -
(20 / num_states),   -2 , 1));
        elseif i == 2
            F1(j, i) = 0.5 * (normcdf(S_t1, 0.8 * S_t + 1 + 0, 1)  -
normcdf(state_range(i-1) - (20/num_states), 0.8 * S_t + 1 , 1));
            F2(j, i) = 0.5 * (normcdf(S_t1, -2 + 0, 1)  -
normcdf(state_range(i-1) - (20/num_states), + -2 , 1));
        elseif i == num_states
            F1(j, i) = 0.5 * (normcdf(S_t1, 0.8 * S_t + 1 , 1)  -
normcdf(state_range(i-1), 0.8 * S_t + 1 , 1));
            F2(j, i) = 0.5 * (normcdf(S_t1,  -2 , 1)    -
normcdf(state_range(i-1),   -2 , 1));
        else
            F1(j, i) = 0.5 * (normcdf(S_t1, 0.8 * S_t + 1 , 1) -
normcdf(state_range(i-2), 0.8 * S_t + 1 + 0, 1));
            F2(j, i) = 0.5 * (normcdf(S_t1,   -2 , 1) ...
                - normcdf(state_range(i-2),   -2 , 1));
        end
    end
end

%% (3) Value Iteration
V1 = zeros(num_states, 1);  % Value function for action 1
V2 = zeros(num_states, 1);  % Value function for action 2

for iter = 1:maxIter
    % Compute updated value functions
```

```
    V1_new = F1 * (R + gamma * max(V1, V2));
    V2_new = F2 * (R + gamma * max(V1, V2));

    % Check convergence
    if max(abs(V1_new - V1)) < tol && max(abs(V2_new - V2)) < tol
        fprintf('Converged in %d iterations.\n', iter);
        break;
    end

    % Update value functions
    V1 = V1_new;
    V2 = V2_new;
end

%% nn
% Define reward function
reward = @(y) min(2, y.^2);

% Parameters
N = 2000;
action = randi([1, 2], 1, N);
set1 = [];
set2 = [];
state = randn;

% Generate state transitions
for i = 1:N
    W = randn;
    if action(i) == 1
        next_state = 0.8 * state + 1 + W;
        set1 = [set1; state, next_state];
    else
        next_state = -2 + W;
        set2 = [set2; state, next_state];
    end
    state = next_state;
end

% Prepare data
X1_data = set1(:, 1)';
Y1_data = set1(:, 2)';
X2_data = set2(:, 1)';
Y2_data = set2(:, 2)';

% Ensure equal lengths of Y1_data and Y2_data by padding
len_diff = abs(length(Y1_data) - length(Y2_data));
if length(Y1_data) < length(Y2_data)
    Y1_data = [Y1_data, nan(1, len_diff)];
elseif length(Y2_data) < length(Y1_data)
    Y2_data = [Y2_data, nan(1, len_diff)];
end

% Training parameters
num_iter = 10000;
lr = 0.0002;  % Reduced learning rate
m = 100;
gamma = 0.8;
loss1 = zeros(1, num_iter);
loss2 = zeros(1, num_iter);
```

```
% Neural network initialization for Network 1
w1 = 2*randn(m, 1) * sqrt(2 / m); % He initialization
b1 = zeros(m, 1);
w2 = 2/m*randn(1, m) * sqrt(2 / m);
b2 = 0;

% Neural network initialization for Network 2
w3 = 2/m*randn(m, 1) * sqrt(2 / m); % He initialization
b3 = zeros(m, 1);
w4 = 2/m*randn(1, m) * sqrt(2 / m);
b4 = 0;

% Activation and other functions
ReLU = @(z) max(0, z);
r = @(y) min(2, y.^2);
phi = @(z) 0.5 * z.^2;
psi = @(z) -z;

% Gradient clipping threshold
grad_clip = 5;

% Training loop
for iter = 1:num_iter
    % Forward pass for Network 1
    Z1 = w1 * X1_data + b1;
    A1 = ReLU(Z1);
    Z2 = w2 * A1 + b2; % Predicted output
    output1 = Z2;

    % Target calculation for Network 1
    z11 = w1 * Y1_data + b1;
    a11 = ReLU(z11);
    om11 = w2 * a11 + b2;

    z12 = w1 * Y2_data + b1;
    a12 = ReLU(z12);
    om12 = w2 * a12 + b2;

    valid_idx1 = ~isnan(Y1_data) & ~isnan(Y2_data);
    YY1 = r(Y1_data(valid_idx1)) + gamma * max(om11(valid_idx1),
om12(valid_idx1));

    % Loss calculation for Network 1
    Z2_valid1 = Z2(valid_idx1);
    c1 = phi(Z2_valid1) + YY1 .* psi(Z2_valid1);
    loss1(iter) = mean(c1);

    % Backpropagation for Network 1
    dZ2_1 = Z2_valid1 - YY1;
    dW2_1 = dZ2_1 * A1(:, valid_idx1)';
    db2_1 = sum(dZ2_1);
    dA1_1 = w2' * dZ2_1;
    dZ1_1 = dA1_1 .* (Z1(:, valid_idx1) > 0);
    dW1_1 = dZ1_1 * X1_data(valid_idx1)';
    db1_1 = sum(dZ1_1, 2);

    % Gradient clipping for Network 1
    dW2_1 = max(min(dW2_1, grad_clip), -grad_clip);
```

```
    db2_1 = max(min(db2_1, grad_clip), -grad_clip);
    dW1_1 = max(min(dW1_1, grad_clip), -grad_clip);
    db1_1 = max(min(db1_1, grad_clip), -grad_clip);

    % Update Network 1
    w2 = w2 - lr * dW2_1;
    b2 = b2 - lr * db2_1;
    w1 = w1 - lr * dW1_1;
    b1 = b1 - lr * db1_1;

    % Forward pass for Network 2
    Z3 = w3 * X2_data + b3;
    A3 = ReLU(Z3);
    Z4 = w4 * A3 + b4; % Predicted output
    output2 = Z4;

    % Target calculation for Network 2
    z21 = w3 * Y2_data + b3;
    a21 = ReLU(z21);
    om21 = w4 * a21 + b4;

    z22 = w3 * Y1_data + b3;
    a22 = ReLU(z22);
    om22 = w4 * a22 + b4;

    valid_idx2 = ~isnan(Y2_data) & ~isnan(Y1_data);
    YY2 = r(Y2_data(valid_idx2)) + gamma * max(om21(valid_idx2),
om22(valid_idx2));

    % Loss calculation for Network 2
    Z4_valid2 = Z4(valid_idx2);
    c2 = phi(Z4_valid2) + YY2 .* psi(Z4_valid2);
    loss2(iter) = mean(c2);

    % Backpropagation for Network 2
    dZ4_2 = Z4_valid2 - YY2;
    dW4_2 = dZ4_2 * A3(:, valid_idx2)';
    db4_2 = sum(dZ4_2);
    dA3_2 = w4' * dZ4_2;
    dZ3_2 = dA3_2 .* (Z3(:, valid_idx2) > 0);
    dW3_2 = dZ3_2 * X2_data(valid_idx2)';
    db3_2 = sum(dZ3_2, 2);

    % Gradient clipping for Network 2
    dW4_2 = max(min(dW4_2, grad_clip), -grad_clip);
    db4_2 = max(min(db4_2, grad_clip), -grad_clip);
    dW3_2 = max(min(dW3_2, grad_clip), -grad_clip);
    db3_2 = max(min(db3_2, grad_clip), -grad_clip);

    % Update Network 2
    w4 = w4 - lr * dW4_2;
    b4 = b4 - lr * db4_2;
    w3 = w3 - lr * dW3_2;
    b3 = b3 - lr * db3_2;
end

x=linspace(-6,6,1000);
Z1 = w1 * x + b1;
A1 = ReLU(Z1);
```

```
Z2 = w2 * A1 + b2; % Predicted output
outputxx1 = Z2;

Z3 = w3 * x + b3;
A3 = ReLU(Z3);
Z4 = w4 * A3 + b4; % Predicted output
outputxx2 = Z4;



% Numerical decision policy
figure; hold on;
policy = ones(num_states, 1); % Default to action 1
policy(V2 > V1) = 2; % Switch to action 2 where V2 > V1
%scatter(state_range, policy, 50, 'b', 'filled'); % Plot numerical policy
plot(state_range, policy, 'k-', 'LineWidth', 2, 'DisplayName', 'Optimal
Policy numerical');
% NN-based decision policy
policyA1 = ones(length(x), 1); % Default to action 1
policyA1(outputxx2 > outputxx1) = 2; % Switch to action 2 where outputxx2 >
outputxx1
%scatter(x, policyA1, 20, 'r', 'filled'); % Plot NN policy
plot(x, policyA1, 'r-', 'LineWidth', 2, 'DisplayName', 'Optimal Policy
[A1]');
% Plot value functions (numerical)
plot(state_range, V1, 'k-', 'LineWidth', 1.5, 'DisplayName', 'V1
(Numerical)');
plot(state_range, V2, 'c-', 'LineWidth', 1.5, 'DisplayName', 'V2
(Numerical)');

% Plot neural network outputs
plot(x, outputxx1, 'r--', 'LineWidth', 2, 'DisplayName', 'NN V1');
plot(x, outputxx2, 'b--', 'LineWidth', 2, 'DisplayName', 'NN V2');

% Formatting
title('Decision Policy and Value Functions');
xlabel('State');
ylabel('Value / Policy');
legend('show');
grid on;
xlim([-5, 5]);
ylim([-6, 13]);
yticks([1, 2]);

% Plot 1: Output after training
figure; hold on;
plot(state_range, V1, 'k-', 'LineWidth', 1.5); hold on; grid on;
plot(state_range, V2, 'c-', 'LineWidth', 1.5); hold on;
plot(x, outputxx1, 'r--', 'LineWidth', 2);
plot(x, outputxx2, 'b--', 'LineWidth', 2);
title('Output After Training');
xlim([-5,5]);
ylim([7,12]);
xlabel('X');
ylabel('Output');

% Plot 2: Loss during training
figure;
plot(1:num_iter, loss1, 'b-', 'LineWidth', 1.5); hold on;
```

```
plot(1:num_iter, loss2, 'r-', 'LineWidth', 1.5);
title('Loss During Training');
xlabel('Iteration');
ylabel('Loss');
legend('Network 1', 'Network 2');
```

Data Driven C1

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% FILE: example_value_function_training.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%

clear; clc; close all;
rng(0);

%% ========================================================================
%  (1)  Build/solve the MDP via Value Iteration on a discrete state space
%       (This part is just to compare with the neural nets)
% ========================================================================
reward = @(s) min(2, s.^2);

% Transition CDFs for two actions
H1 = @(st1, st) normcdf(st1, 0.8*st + 1, 1);  % Action 1
H2 = @(st1, st) normcdf(st1, -2,        1);  % Action 2

gamma = 0.8;   % Discount factor
tol   = 1e-6;  % Convergence tolerance
maxIter = 1000;

% Discretize State Space
num_states   = 500;
state_range  = linspace(-6, 6, num_states);  % S in [-6, 6]
R_discrete   = reward(state_range)';         % reward for each discrete
state

% Build transition matrices F1, F2
F1 = zeros(num_states, num_states);
F2 = zeros(num_states, num_states);
for j = 1:num_states
    s_now = state_range(j);
    for i = 1:num_states
        s_next = state_range(i);

        % We'll do a simplistic approach: difference of CDFs
        % For subinterval boundaries, you can adjust carefully
        if i == 1
            F1(j,i) = H1(s_next, s_now) - H1(s_next - (12/num_states),
s_now);
            F2(j,i) = H2(s_next, s_now) - H2(s_next - (12/num_states),
s_now);
        else
            % Approx edges similarly. This is just a rough example.
            left_boundary  = state_range(i-1);
            p1_left  = H1(left_boundary, s_now);
            p2_left  = H2(left_boundary, s_now);
            p1_right = H1(s_next,       s_now);
            p2_right = H2(s_next,       s_now);
```

```
            F1(j,i) = p1_right - p1_left;
            F2(j,i) = p2_right - p2_left;
        end
    end
end

% Value Iteration
V1 = zeros(num_states,1);
V2 = zeros(num_states,1);
for iter = 1:maxIter
    V1_new = F1 * (R_discrete + gamma*max(V1,V2));
    V2_new = F2 * (R_discrete + gamma*max(V1,V2));

    if max(abs(V1_new - V1)) < tol && max(abs(V2_new - V2)) < tol
        fprintf('Value Iteration converged in %d steps.\n', iter);
        break;
    end
    V1 = V1_new;
    V2 = V2_new;
end

%% ========================================================================
%  (2)  Generate Random Transitions (continuous) & Extract Rewards
% ========================================================================
N = 2000;                           % Number of transitions total
action = randi([1, 2], 1, N);       % Randomly pick action 1 or 2
set1 = [];  % (state, next_state) pairs for action=1
set2 = [];  % (state, next_state) pairs for action=2

% Start from some random state
state = randn;
for i = 1:N
    noise_w = randn;   % W ~ N(0,1)
    if action(i) == 1
        next_state = 0.8*state + 1 + noise_w;
        set1 = [set1; state, next_state];
    else
        next_state = -2 + noise_w;
        set2 = [set2; state, next_state];
    end
    state = next_state;
end

% Convert to row vectors for convenience
X1_data = set1(:,1)';   % states for action=1
Y1_data = set1(:,2)';   % next states for action=1
X2_data = set2(:,1)';   % states for action=2
Y2_data = set2(:,2)';   % next states for action=2

% Compute immediate rewards for each next state
R1_data = reward(Y1_data);
R2_data = reward(Y2_data);

% We might have different lengths in set1 vs set2
% -> pad the shorter with NaNs
len1 = length(Y1_data);
len2 = length(Y2_data);
len_diff = abs(len1 - len2);
```

```
if len1 < len2
    % pad set1 with NaNs at the end
    X1_data = [X1_data, nan(1, len_diff)];
    Y1_data = [Y1_data, nan(1, len_diff)];
    R1_data = [R1_data, nan(1, len_diff)];
elseif len2 < len1
    % pad set2 with NaNs
    X2_data = [X2_data, nan(1, len_diff)];
    Y2_data = [Y2_data, nan(1, len_diff)];
    R2_data = [R2_data, nan(1, len_diff)];
end

%% ======================================================================
%  (3)  Neural Network Setup
% ======================================================================
numIterations = 10000;  % training iterations
learningRate  = 2e-4;   % mu
momentum      = 0.99;   % lambda
stabConst     = 0.1;    % c
numHidden     = 100;    % # hidden neurons
a = 0;
b = 10;   % final activation scaling

% Network 1 params
A01 =  2*randn(numHidden,1)*sqrt(2/numHidden);
B01 =  zeros(numHidden,1);
A11 = (2/numHidden)*randn(numHidden,1)*sqrt(2/numHidden);
B11 = -0.3;

% Momentum accumulators
PDA01 = zeros(numHidden,1);
PDB01 = zeros(numHidden,1);
PDA11 = zeros(numHidden,1);
PDB11 = 0;

% Network 2 params
A02 = (2/numHidden)*randn(numHidden,1)*sqrt(2/numHidden);
B02 = zeros(numHidden,1);
A12 = (2/numHidden)*randn(numHidden,1)*sqrt(2/numHidden);
B12 = 0;

% Momentum accumulators
PDA02 = zeros(numHidden,1);
PDB02 = zeros(numHidden,1);
PDA12 = zeros(numHidden,1);
PDB12 = 0;

% For logging cost
costNet1 = zeros(numIterations,1);
costNet2 = zeros(numIterations,1);

%% ======================================================================
%  (4)  Training Loop
% ======================================================================
for iter = 1:numIterations

    %% ------------------- NETWORK 1: FORWARD PASS -----------------------
    % "current state" for action=1
    Z1_cur = bsxfun(@plus, A01*X1_data, B01);  % size: (numHidden x len1)
```

```
    X1_cur = max(Z1_cur, 0);                    % ReLU
    W1_cur = A11' * X1_cur + B11;               % (1 x len1)
    Q1_cur = a./(1+exp(W1_cur)) + b./(1+exp(-W1_cur));  % final activation
=> Q1(s)

    % "next state" evaluation => Q1(next_s) and Q2(next_s), for each sample
    Z1_nextSt = bsxfun(@plus, A01*Y1_data, B01);
    X1_nextSt = max(Z1_nextSt, 0);
    W1_nextSt = A11' * X1_nextSt + B11;
    Q1_nextSt_1 = a./(1+exp(W1_nextSt)) + b./(1+exp(-W1_nextSt));  % Q1(
next_s )

    Z1_nextSt_2 = bsxfun(@plus, A02*Y1_data, B02);
    X1_nextSt_2 = max(Z1_nextSt_2, 0);
    W1_nextSt_2 = A12' * X1_nextSt_2 + B12;
    Q1_nextSt_2 = a./(1+exp(W1_nextSt_2)) + b./(1+exp(-W1_nextSt_2)); % Q2(
next_s )

    % Valid samples for network 1 (ignore NaNs from padding)
    validMask1 = ~isnan(X1_data) & ~isnan(Y1_data) & ~isnan(R1_data);

    % Slice out relevant columns for training
    Q1_cur_valid     = Q1_cur(validMask1);
    Q1_ns_1_valid    = Q1_nextSt_1(validMask1);
    Q1_ns_2_valid    = Q1_nextSt_2(validMask1);
    R1_valid         = R1_data(validMask1);

    % Build the target for network 1
    Y1_target = R1_valid + gamma .* max(Q1_ns_1_valid, Q1_ns_2_valid);

    % Also gather hidden activations to backprop
    Z1_cur_valid = Z1_cur(:, validMask1);  % (numHidden x #valid)
    X1_cur_valid = X1_cur(:, validMask1);
    W1_cur_valid = W1_cur(validMask1);
    maskZ1_cur_valid = double(Z1_cur_valid > 0);

    % Convert to column vectors for convenience
    W1_col     = W1_cur_valid(:);
    Q1_col     = Q1_cur_valid(:);
    T1_col     = Y1_target(:);

    % d/dW = -(1/(1+exp(-W))) * (Target - Output)
    deltaOutput1 = -(1./(1+exp(-W1_col))) .* (T1_col - Q1_col);  % (#valid
x 1)

    % Gradient wrt A11, B11
    dA11 = X1_cur_valid * deltaOutput1;   % (numHidden x 1)
    dB11 = sum(deltaOutput1);

    % For A01, B01 we also need X1_data(valid)
    X1_data_valid = X1_data(validMask1);
    X1_data_mat   = repmat(X1_data_valid, numHidden, 1);  % (numHidden x
#valid)
    mask_mul_1    = maskZ1_cur_valid .* X1_data_mat;
    dA01_vec      = mask_mul_1 * deltaOutput1;             % (numHidden x
1)
    dA01          = A11 .* dA01_vec;  % as per your custom formula

    dB01_vec = maskZ1_cur_valid * deltaOutput1;  % (numHidden x 1)
```

```
    dB01     = A11 .* dB01_vec;

    % Momentum accumulators for net 1
    PDA11 = momentum * PDA11 + (1 - momentum)*(dA11.^2);
    PDB11 = momentum * PDB11 + (1 - momentum)*(dB11.^2);
    PDA01 = momentum * PDA01 + (1 - momentum)*(dA01.^2);
    PDB01 = momentum * PDB01 + (1 - momentum)*(dB01.^2);

    % Update
    A01 = A01 - learningRate*(dA01 );
    B01 = B01 - learningRate*(dB01);
    A11 = A11 - learningRate*(dA11 );
    B11 = B11 - learningRate*(dB11 );

    %% ------------------- NETWORK 2: FORWARD PASS -----------------------
    Z2_cur = bsxfun(@plus, A02*X2_data, B02);
    X2_cur = max(Z2_cur, 0);
    W2_cur = A12' * X2_cur + B12;
    Q2_cur = a./(1+exp(W2_cur)) + b./(1+exp(-W2_cur));

    % Next-state
    Z2_nextSt = bsxfun(@plus, A02*Y2_data, B02);
    X2_nextSt = max(Z2_nextSt, 0);
    W2_nextSt = A12' * X2_nextSt + B12;
    Q2_nextSt_2 = a./(1+exp(W2_nextSt)) + b./(1+exp(-W2_nextSt));  %
Q2(next_s)

    Z2_nextSt_1 = bsxfun(@plus, A01*Y2_data, B01);
    X2_nextSt_1 = max(Z2_nextSt_1, 0);
    W2_nextSt_1 = A11' * X2_nextSt_1 + B11;
    Q2_nextSt_1 = a./(1+exp(W2_nextSt_1)) + b./(1+exp(-W2_nextSt_1));  %
Q1(next_s)

    validMask2 = ~isnan(X2_data) & ~isnan(Y2_data) & ~isnan(R2_data);

    Q2_cur_valid     = Q2_cur(validMask2);
    Q2_ns_2_valid    = Q2_nextSt_2(validMask2);
    Q2_ns_1_valid    = Q2_nextSt_1(validMask2);
    R2_valid         = R2_data(validMask2);

    % Target for network2
    Y2_target = R2_valid + gamma .* max(Q2_ns_1_valid, Q2_ns_2_valid);

    % For backprop
    Z2_cur_valid  = Z2_cur(:, validMask2);
    X2_cur_valid  = X2_cur(:, validMask2);
    W2_cur_valid  = W2_cur(validMask2);
    maskZ2_cur_valid = double(Z2_cur_valid > 0);

    W2_col  = W2_cur_valid(:);
    Q2_col  = Q2_cur_valid(:);
    T2_col  = Y2_target(:);

    deltaOutput2 = -(1./(1+exp(-W2_col))) .* (T2_col - Q2_col);

    dA12 = X2_cur_valid * deltaOutput2;
    dB12 = sum(deltaOutput2);

    X2_data_valid = X2_data(validMask2);
```

```
    X2_mat        = repmat(X2_data_valid, numHidden, 1);
    mask_mul_2    = maskZ2_cur_valid .* X2_mat;
    dA02_vec      = mask_mul_2 * deltaOutput2;
    dA02          = A12 .* dA02_vec;

    dB02_vec = maskZ2_cur_valid * deltaOutput2;
    dB02     = A12 .* dB02_vec;

    % Momentum accumulators for net 2
    PDA12 = momentum * PDA12 + (1 - momentum)*(dA12.^2);
    PDB12 = momentum * PDB12 + (1 - momentum)*(dB12.^2);
    PDA02 = momentum * PDA02 + (1 - momentum)*(dA02.^2);
    PDB02 = momentum * PDB02 + (1 - momentum)*(dB02.^2);

    % Update
    A02 = A02 - learningRate*(dA02 ./ sqrt(stabConst + PDA02));
    B02 = B02 - learningRate*(dB02 ./ sqrt(stabConst + PDB02));
    A12 = A12 - learningRate*(dA12 ./ sqrt(stabConst + PDA12));
    B12 = B12 - learningRate*(dB12 ./ sqrt(stabConst + PDB12));

    %% ---------------------------------------------------------------
    %  Compute a simple cost measure for logging
    %% ---------------------------------------------------------------
    costNet1(iter) = 0.5 * mean((T1_col - Q1_col).^2, 'omitnan');
    costNet2(iter) = 0.5 * mean((T2_col - Q2_col).^2, 'omitnan');
end

fprintf('Training complete. Networks updated.\n');

%% ========================================================================
%  (5)  Plot the Loss
% ========================================================================
figure;
plot(1:numIterations, costNet1, 'b-', 'LineWidth',1.5); hold on;
plot(1:numIterations, costNet2, 'r-', 'LineWidth',1.5);
xlabel('Iteration');
ylabel('Loss');
legend('Network 1','Network 2');
title('Loss During Training');
grid on;

%% ========================================================================
%  (6)  Evaluate final networks over a grid y in [-6,6]
% ========================================================================
y = linspace(-6,6,1000);

% Network 1
Z1_eval = bsxfun(@plus, A01*y, B01);
X1_eval = max(Z1_eval,0);
W1_eval = A11'*X1_eval + B11;
Q1_eval = a./(1+exp(W1_eval)) + b./(1+exp(-W1_eval));

% Network 2
Z2_eval = bsxfun(@plus, A02*y, B02);
X2_eval = max(Z2_eval,0);
W2_eval = A12'*X2_eval + B12;
Q2_eval = a./(1+exp(W2_eval)) + b./(1+exp(-W2_eval));

figure; hold on;
```

```
plot(state_range, V1, 'k-','LineWidth',1.5, 'DisplayName','V1 (Action1,
MDP)');
plot(state_range, V2, 'k-','LineWidth',1.5, 'DisplayName','V2 (Action2,
MDP)');
plot(y, Q1_eval, 'g--','LineWidth',2, 'DisplayName','NN - Q1');
plot(y, Q2_eval, 'm--','LineWidth',2, 'DisplayName','NN - Q2');
xlabel('State S'); ylabel('Network Output');
title('Final Networks Output vs. Value Iteration');
legend('Location','best'); grid on;

fprintf('All done!\n');

%% ========================================================================
%  (7)  Plot Optimal Decision Policy
% ========================================================================

% From Value Iteration (numerical)
policy_numerical = ones(num_states, 1); % Default action 1
policy_numerical(V2 > V1) = 2;          % Action 2 where V2 > V1

% From Neural Network
policy_nn = ones(size(y)); % Default action 1
policy_nn(Q2_eval > Q1_eval) = 2; % Action 2 where Q2_eval > Q1_eval

% Plotting
figure; hold on;
plot(state_range, V1, 'k-','LineWidth',1.5, 'DisplayName','V1');
plot(state_range, V2, 'k-','LineWidth',1.5, 'DisplayName','V2');
plot(y, Q1_eval, 'g--','LineWidth',2, 'DisplayName','Network1');
plot(y, Q2_eval, 'm--','LineWidth',2, 'DisplayName','Network2');
plot(state_range, policy_numerical, 'k-', 'LineWidth', 2, 'DisplayName',
'Optimal Policy numerical');
plot(y, policy_nn, 'r-', 'LineWidth', 2, 'DisplayName', 'Optimal Policy
[C1]');

% Formatting
xlim([-5,5]);
ylim([-6, 13]);
yticks([1, 2]);
xlabel('State S');
ylabel('Optimal Action');
title('Optimal Decision Policy');
legend('Location', 'best');
grid on;

fprintf('Optimal Decision Policy plotted.\n');
```