Εργασία 3:

Υπολογισμός υπό συνθήκη μέσων τιμών(conditional expectations) με αριθμητικές και data driven τεχνικές και χρήση τους για myopic και infinite horizon reinforcement learning.

Λάμπρος Αβούρης (1092732)

Πανεπιστήμιο Πατρών

Μηχανική μάθηση

Διδάσκων: Γ. Μουστακίδης

Άσκηση 3.1

Μας ζητείται η υπολογίσουμε δύο υπό συνθήκη μέσες τιμές :

$$\mathbb{E}[Y \mid X = X], \quad \mathbb{E}\left[\min\{1, \max\{-1, Y\}\} \mid X = X\right]$$

Που αντιστοιχούν σε συναρτήσεις

$$G(Y) = Y, \quad G(Y) = \min\{1, \max\{-1, Y\}\}$$

Με

$$Y = 0.8X + W\, Y = 0.8X + W\, Y = 0.8X + W$$

Για τον υπολογισμό αυτό αρκεί να υπολογίσω το ολοκλήρωμα

$$U = \int G \, dH(y \mid x)$$

Οπου H η υπο συνθηκη πυκνότητα πιθανότητας. G η συνάρτηση.

Αρχικά καλούμαστε να αποδείξουμε ότι η

$$\mathbb{E}\left[\min\{1, \max\{-1, Y\}\} \mid X = X\right]$$

Είναι bounded στο [-1,1],

Αυτο εύκολα αποδεικνύεται καθώς:

Αν μια οποιαδήποτε τυχαία μεταβλητή W ισχύε:

$$\alpha \leq W \leq \beta \implies \alpha \leq \mathbb{E}(W) \leq \beta$$

Θέτοντας

$$W = \min\{1, \max\{-1, Y\}\}$$

$$-1 \leq W \leq 1 \implies -1 \leq \mathbb{E}[W] \leq 1 \implies -1 \leq \mathbb{E}[\min\{1, \max\{-1, Y\}\}] \leq 1$$

Ο υπολογισμός του ολοκληρώματος θα επιτευχθεί αριθμητικά χρησιμοποιώντας τον τύπο για trapezoidal integration που μας δόθηκε στην lecture 11 pg 4

Δεν θα κάνουμε μετακίνηση όρων ώστε να μπορούμε να χρησιμοποιήσουμε άμεσα τις λειτουργίες του numpy για τον υπολογισμό των διαφορών.

Θα κάνουμε την ίδια διαδικασία και στις δύο περιπτώσεις,αλλάζοντας μόνο την G(Y)

Για την Data driven implementation θα δημιουργήσουμε 4 νευρωνικά δίκτυα, ένα για κάθε κριτήριο. Για την εκπαίδευση των νευρωνικών δικτύων, σύμφωνα με την θεωρία που παρουσιάζεται στο δεδομένο paper, Data-Driven Estimation of Conditional Expectations, Application to Optimal Stopping and Reinforcement Learning (George V. Moustakides) ,καθώς και στις σημειώσεις lecture 11 pg 13, πρέπει να ελαχιστοποιήσω την συνάρτηση

$$J = \mathbb{E}\left[\varphi + G(y)\psi\right]$$

Τα νευρωνικά δίκτυα θα εκτιμούν τις μέσες τιμές υ, αν εφαρμόσουμε την έξοδό τους καταλληλη συναρτηση ω , τις οποίες πριν υπολογίζαμε αριθμητικά.

Οπου φ ,ψ και ω οι συναρτήσεις που μας δίνονται στα κριτήρια.

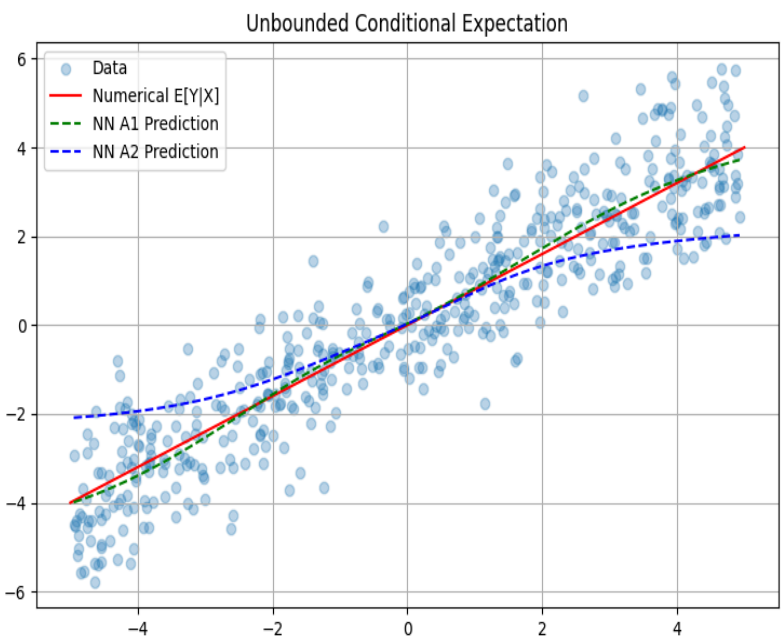
Τα κριτήρια που χρησιμοποιούμε είναι τα A1 A2 C1 που αναλύονται στο paper

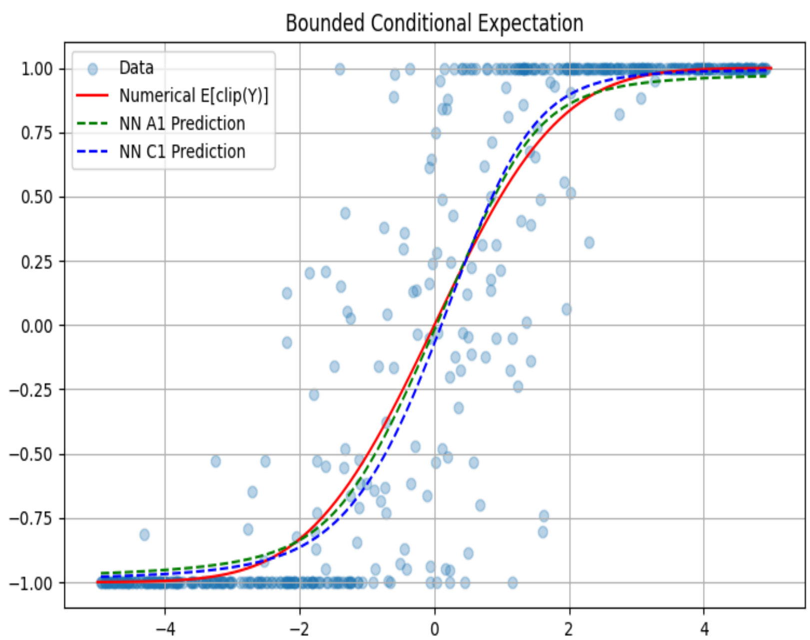
Θα δημιουργήσουμε random data pairs όπως ζητά η εκφώνηση χρησιμοποιώντας την pdf που μας δίνεται για Y , ενώ για X θα πάρω uniform σε συγκεκριμένο διάστημα.

Για τα neural nets θα χρησιμοποιήσω την βιβλιοθήκη torch, εφαρμόζοντας gradient descent, και κάνοντας ADAM optimization όπως προτείνει το paper.

Αφού κάνω training τα δίκτυα βάσει των παραπάνω, θα πάρω την έξοδο τους και θα την εφαρμόσω στην συνάρτηση ω, ώστε να πάρω την τελική πρόβλεψη.

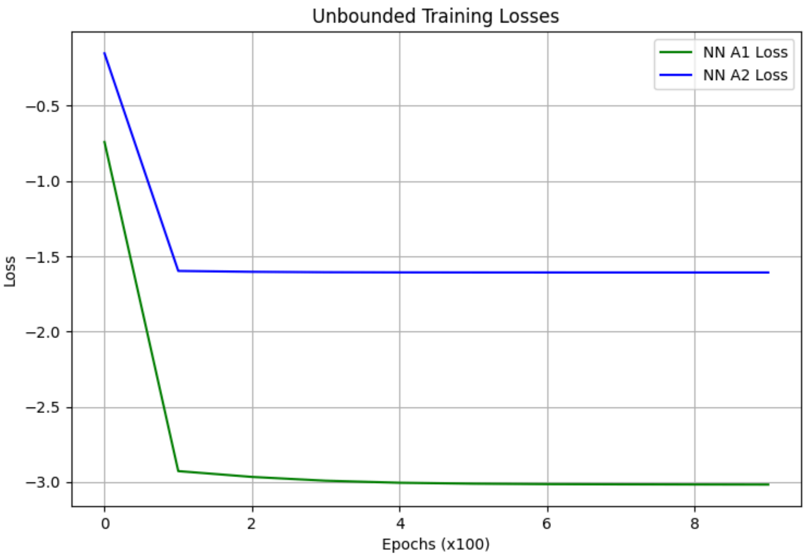
Παίρνω τελικά τα εξής αποτελέσματα:

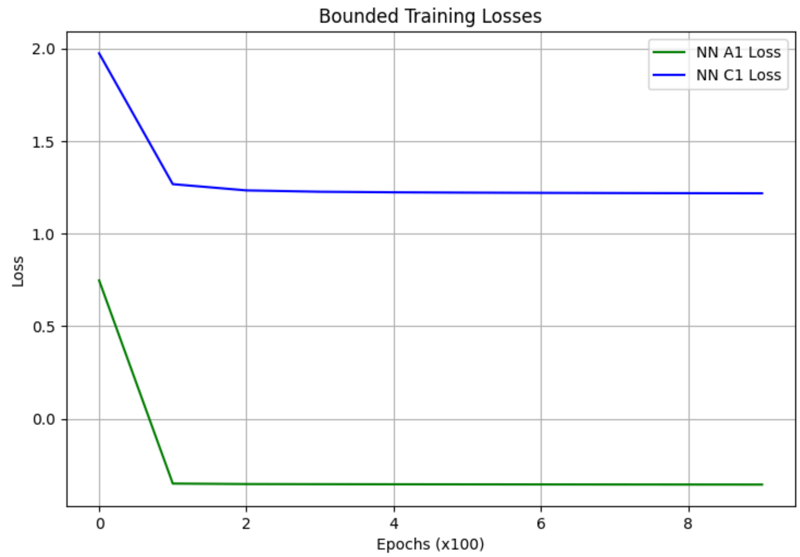




Παρατηρώ πολύ καλά αποτελέσματα για τις μεθόδους A1 C1 αλλά όχι τόσο καλά για την A2.
Παρατηρώ επίσης ότι η unbounded expectation έχει μια σταθερή συγκεκριμένη κλιση 0.8 στο R.
Η bounded conditional expectation παρατηρώ ότι προσεγγίζει μια σταθερή κλίση κοντά στο 0 αλλα γίνεται saturate στα bounds -1 1 κατι το οποίο είναι αναμενόμενο.
Και οι δύο καμπύλες είναι συμμετρικές γύρω του μηδενός.

Τα losses έως να έχω convergence





Άσκηση 3.2 και 3.3

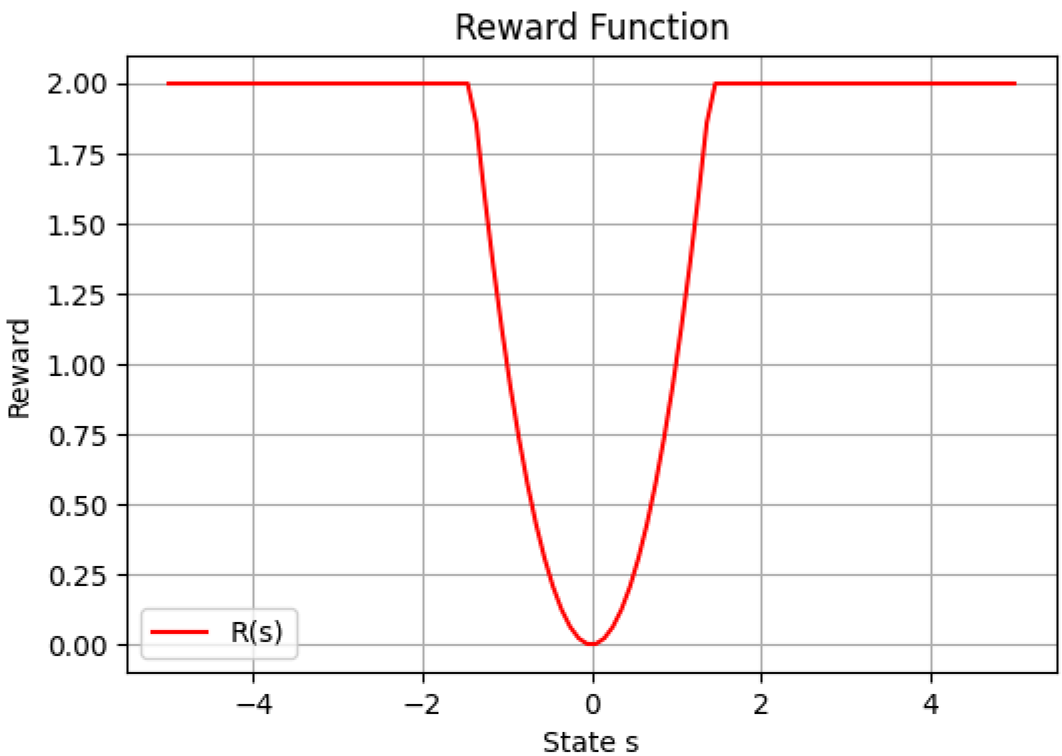
Εδώ θέλουμε να σχεδιάσουμε ένα decision policy για έναν agent ο οποίος βρίσκεται σε καταστάσεις S που προκύπτουν απο Markov decision process με τις εξής μεταβάσεις:

$$\text{For } \alpha = 1 : \quad S_{t+1} = 0.8S_t + 1.0 + W$$

$$\text{For } \alpha = 2 : \quad S_{t+1} = -2.0 + W_t$$

Οπου θα έχω το εξής reward function :

$$R(S) = \min\{2, S^2\}$$



Πρακτικά θέλουμε ο agent να αποφεύγει την περιοχή minimum reward $-\sqrt{2}, \sqrt{2}$

Η Policy 1 αποτελεί ουσιαστικά ομαλή λειτουργία ενώ η Policy 2 αποτελεί reset, για την περίπτωση που η Policy 1 οδηγεί τον agent να μένει στην περιοχή.

Ασκηση 3.2

Στην προκειμένη περίπτωση θα χρησιμοποιήσουμε short sighted reinforcement learning έτσι ώστε να μπορέσουμε να υπολογίσουμε το expected reward δεδομένου S και να μπορούμε να λάβουμε απόφαση.

Καθώς έχω short sighted learning η εξίσωση για το expected reward είναι η εξης

$$V(s) \approx \mathbb{E}[R_0 \mid S_0 = s]$$

Άρα πρέπει να υπολογίσω αυτο το conditional expectation

Καθώς το conditional expectation προκύπτει απο bounded συνάρτηση 0< R(S)<2

Είναι και αυτό bounded στο 0,2 όπως δείξαμε και παραπάνω.

Η αριθμητική μέθοδος είναι παρόμοια με αυτή που δείξαμε παραπάνω ώστε να υπολογίσω expected reward για action 1, v1 και για action 2 v2.

Χρησιμοποιουμε trapezoidal integration για να βρω τα v1 και v2 και τα συγκρίνω επιλέγοντας το μεγαλύτερο, ώστε να έχουμε numerically optimal action policy.

Για το data driven κομματι, θα εφαρμόσουμε minimization της J συνάρτησης που είπαμε πριν

$$J = \mathbb{E}\left[\varphi + G(y)\psi\right]$$

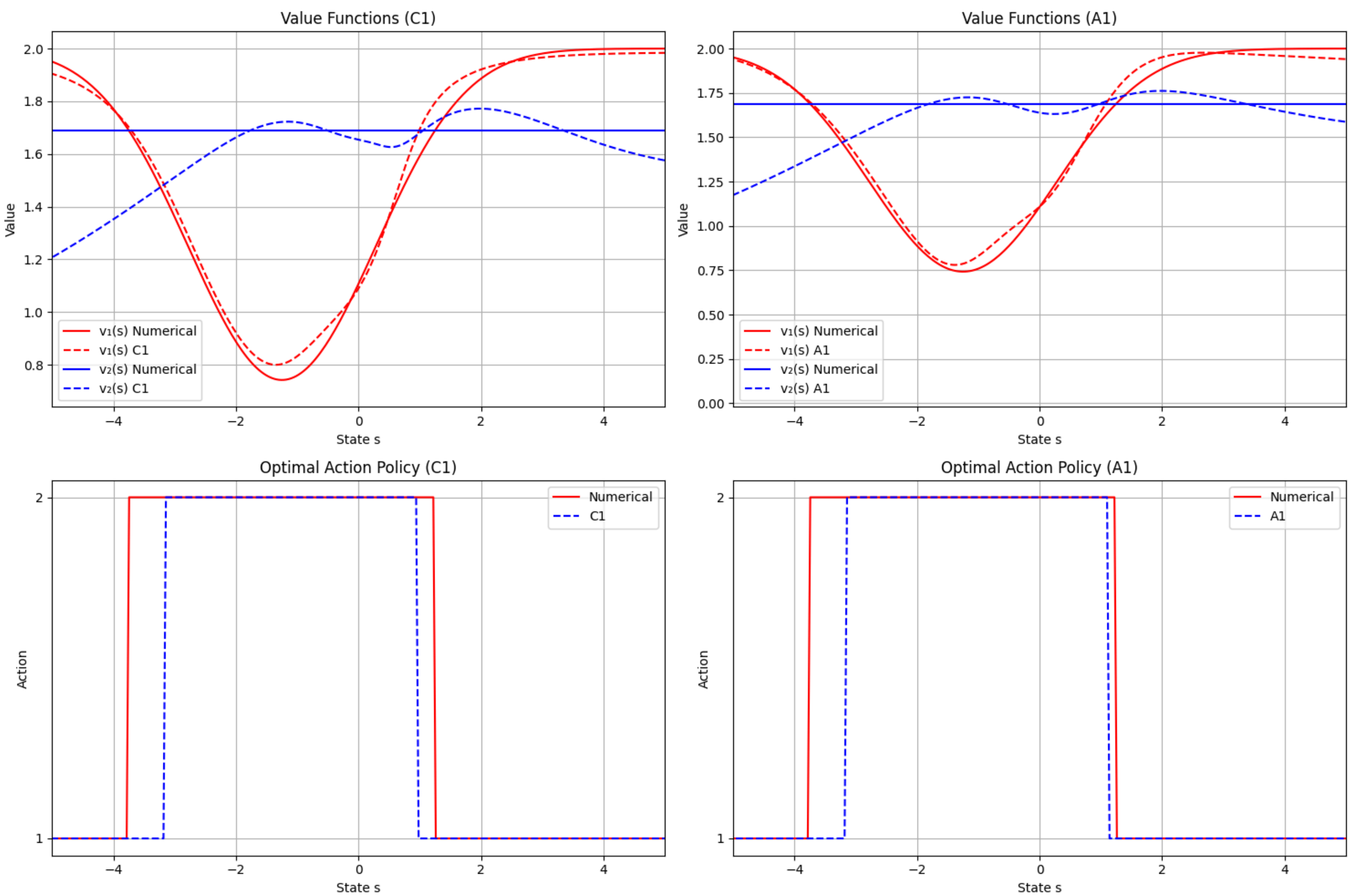
Οταν κανουμε minimize θα μπορώ να πάρω τις v1, v2 εφαρμόζοντας την εξοδο των δικτύων στην ω.

Οι φ,ψ,ω είναι αυτές που βρισκονται στο paper.

Για να κανω training θα δημιουργήσω 1001 samples όπως είπαμε στην εκφώνηση, παιρνοντας 1000 random actions απο random initial state. Θα κανουμε train το neural net με τα a1 και a2 actions ώστε να μπορούμε να κάνουμε predictions και θα επιλέξουμε αναλόγως την action με το μέγιστο reward για να φτιάξουμε την optimal action policy.

Θα χρησιμοποιήσω gradient descent με ADAM optimization όπως προτείνει το paper.

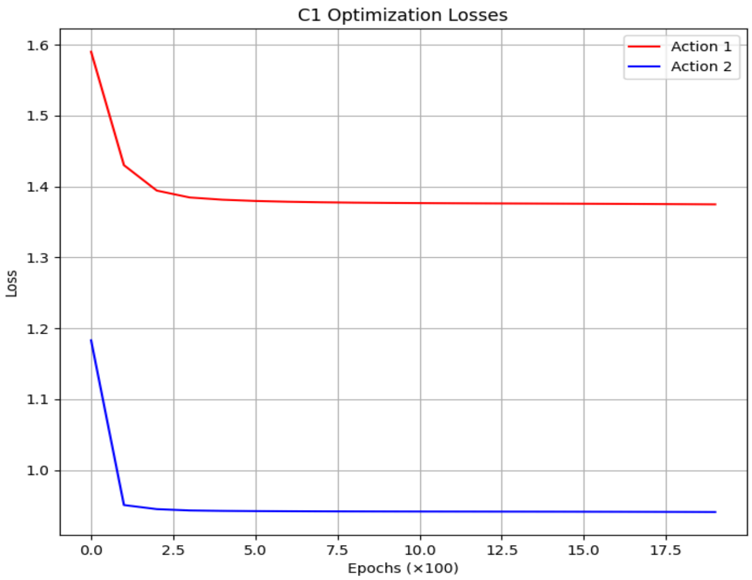
Παιρνω τα εξής αποτελέσματα.:

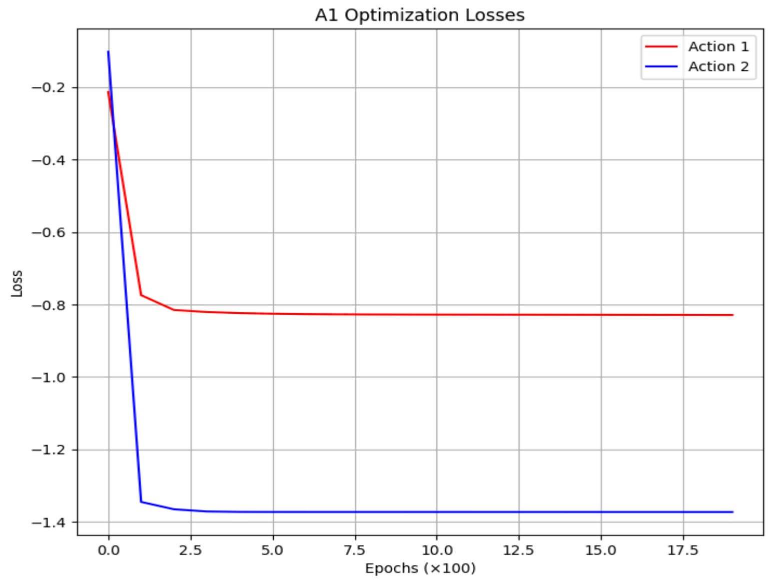


- Παρατηρώ ότι μπορώ να έχω μια πολύ καλή προσέγγιση της action policy μεσω του neural net.
- Επίσης φαίνεται ότι η προσέγγιση είναι χειρότερη για μικρά s απο ότι για μεγαλύτερα.

- Κοιτώντας την reward function μπορώ να παρατηρήσω ότι, optimal action policy είναι ουσιαστικά εκείνη που θα μας κρατήσει έξω από την περιοχή minimum reward $-\sqrt{2}, \sqrt{2}$
- Παρατηρώ οτι για μεγιστοποίηση του reward, γενικά καλύτερη είναι η action policy 1 για όλα τα state αναμεσα σε -4 και 1. Ενώ η policy 2 μεγιστοποιεί το reward εκτός της περιοχής αυτής. Δηλαδή αυτές οι policies είναι πιθανότερο να μας κρατήσουν εκτός του minimum reward περιοχης δεδομένου ότι ήδη βρισκόμαστε σε ένα συγκεκριμένο state.
- Ενώ εκτός της περιοχής αυτής παρατηρώ οτι υπάρχει μόνο policy 1, αυτό διότι, η policy 1 σε εκείνη την περιοχη, οδηγεί το absolute state να αυξανεται η να μειώνετε, αρα πάντα ξεφεύγει απο την περιοχη minimum reward
- Υπάρχει μια μικρή απόκλιση της data driven λύσης, σε σχέση με την αριθμητική λύση, όμως καθώς αυτή είναι μακριά απο την minimum reward περιοχή, δύσκολο να μας οδηγήσει στο να κολλήσουμε εκεί.

Τα losses





Άσκηση 3.3

Εδώ θα επαναλάβω τη ίδια διαδικασία όμως θα χρησιμοποιήσω long sighted reinforcement learning η εξίσωση τώρα γίνεται:

$$V(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t \mid s_0 = s\right]$$

Άρα πρέπει να υπολογίσω και τα rewards για τις επόμενες πράξεις. Όμως η expectation παραπάνω είναι απειρη, άρα πρέπει να βρώ τρόπο ώστε να την υπολογίσω.

Με την αριθμητική μέθοδο, αυτο θα επιτευχθεί εφαρμόζοντας τις εξισώσεις που υπάρχουν στην lecture 12, και κάνοντας τους απαραίτητους υπολογισμούς μέχρι να έχω σύγκλιση.

Για τα neural nets χρησιμοποιώ τα ίδια κριτήρια και διαδικασία, όμως αυτή την φορά κάνω τις απαραίτητες τροποποιήσεις για infinite reward όπως παρουσιάζονται στην lecture 12 16-20

Θέτοντας

$$v_j(X) = \mathbb{E}^j_{S_{t+1}}\left[\mathcal{R}(S_{t+1}) + \gamma Q(S_{t+1}) \mid S_t = X\right]$$

Οπου

$$Q(X) = \max\{v_1(X), \dots, v_K(X)\}$$

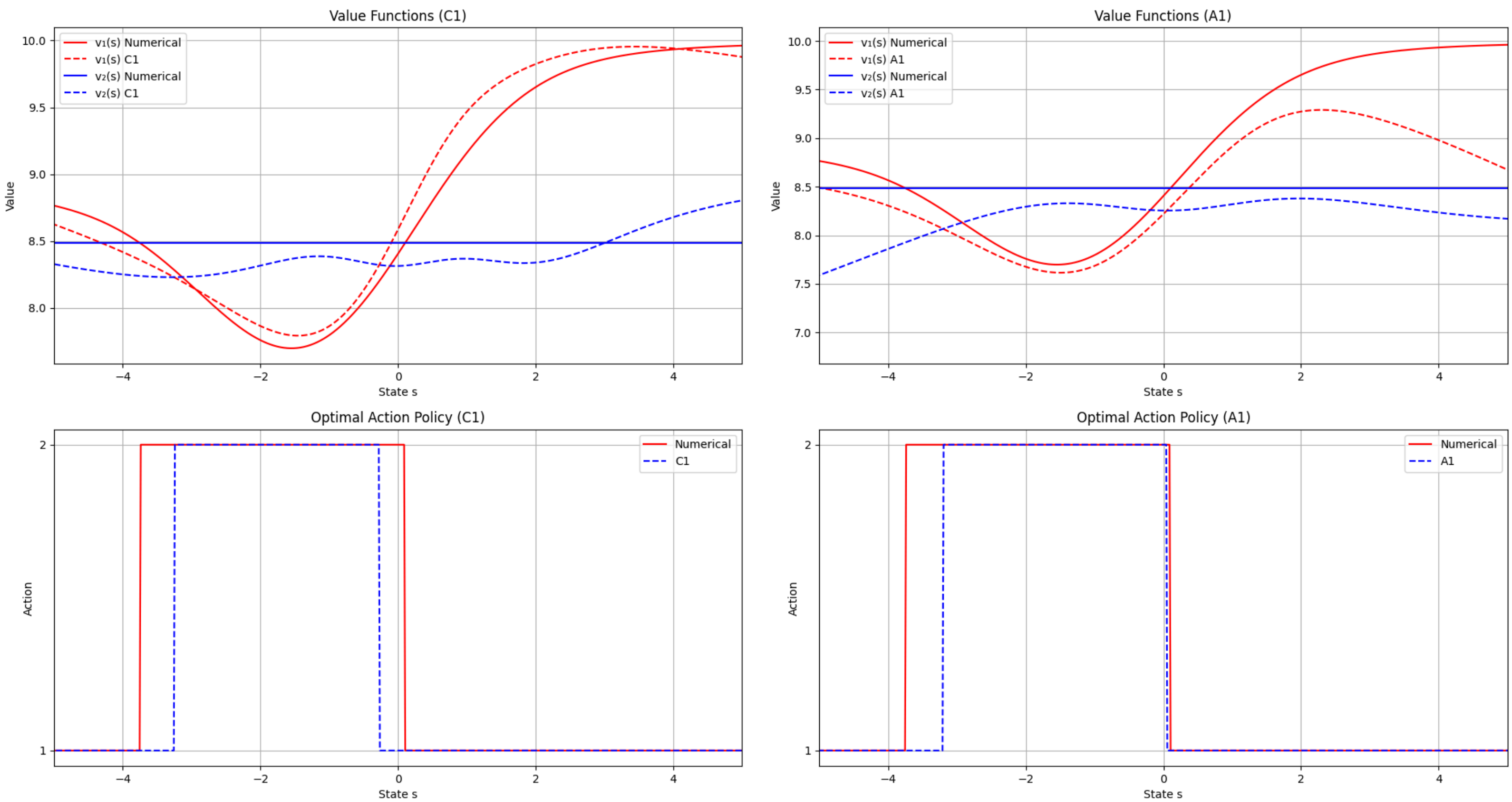
Άρα απλά θα πρέπει να ορίσω νέες estimations κατα το training, χρησιμοποιώντας και την παρούσα αλλα και τις μελλοντικές καταστάσεις.

Πρέπει επίσης να υπολογίσω τα νέα bounds.

Απο τον τύπο που μας δίνεται στην lecture 12 και έχοντας ήδη αποδείjei oti h R(S) είναι bounded εύκολα βρίσκο d1 =0 d2 = 2/(1-0.8) = 10

Κατα τα αλλα, η υλοποίηση είναι παρόμοια.

Παίρνω τα εξής αποτελέσματα,



Παρατηρώ τα εξής:

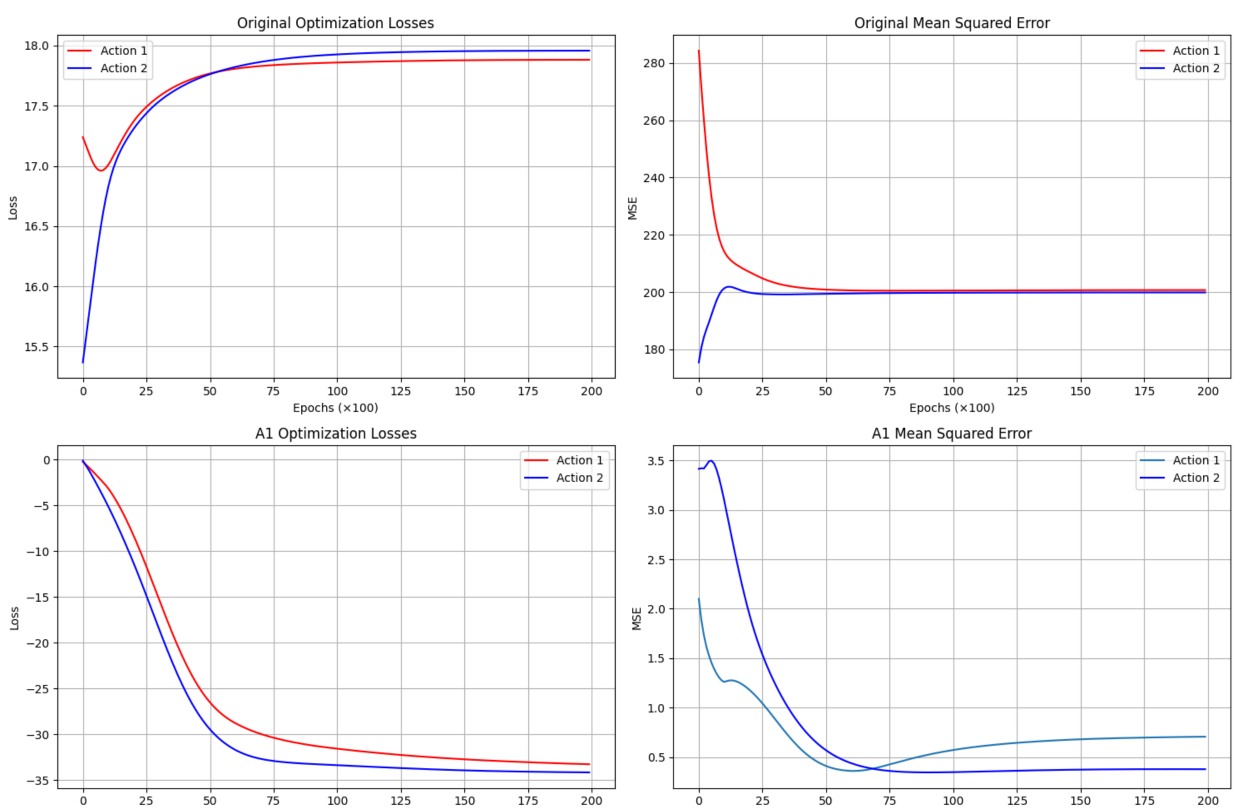
- Αρχικά το value function προφανώς παίρνει πολύ μεγαλύτερες τιμές, αυτό είναι εξαιτίας
- Του νέου ορο $\gamma * Q$ που τοποθετήσαμε.
- Επίσης παρατηρώ οτι η optimal action policy είναι παρόμοια με πριν όμως η περιοχή action 2 είναι στενότερη καθώς η action 2 είναι σε περιοχή -3.8 - 0 πλεον.

  Θυμίζουμε ότι, optimal action policy είναι ουσιαστικά εκείνη που θα μας κρατήσει έξω από την περιοχή minimum reward .
- Αρα έχοντας αυτό δεδομένο και βλέποντας ότι η περιοχή a2(reset decision) πλεον είναι στενότερη,αυτό διότι πλέον δεν λαμβάνεται υπόψη μόνο το επόμενο state, αλλα και τα

επόμενα states μπορούμε να πούμε ότι η προηγούμενη πολιτική θα μας οδηγούσε σε εσφαλμένες απόφασης α2 αν βρισκόμασταν στην περιοχη 0,1 κατι που σημαινει οτι ο agent θα λάμβανε αρκετα μικρότερο reward καθώς μάλιστα το 0,1 βρίσκεται στην περιοχη minimum reward.

- Αυτό επί της ουσίας σημαίνει οτι αν και η α2 έχει το μεγαλύτερο άμεσο reward στην περιοχή 0,1 θα έχει μικρότερο μακροπρόθεσμο reward.
- Παρατηρώ επίσης ότι τα neural nets είχαν ελαφρός μικρότερες αποκλίσεις από πριν απο την αριθμητική τιμή
- Ουσιαστικά αυτή η μέθοδος αποκαλύπτει, οτι η Policy1 τείνει να απομακρύνει το state από την περιοχή minimum reward σε περισσότερα states από ότι στην αρχή είδαμε

Losses



Εδώ το loss c1 για κάποιο λόγο συγκλίνει ανοδικά, δεν εχω καταλαβει ακριβως ποιό είναι το λάθος όμως η εκπαίδευση του neural net δουλεύει.

Έχω τοποθετήσει και το MSE error απο δίπλα για να δείξω ότι όντως μειώνεται

Γενικές παρατηρήσεις :

- Για το συγκεκριμένο σενάριο, η infinite horizon reinforced learning method δεν προσφέρει πολύ μεγάλο πλεονέκτημα, όμως αν είχαμε περισότερα states, ή ακομα σημαντικότερα πιο περίπλοκο reward function τότε θα φαίνεται περισσότερο η χρησημότητα της.
- Η infinite horizon reinforced learning method είναι αρκετά περισσότερο κοστοβόρα, απο αποψη και χρόνου και πόρων.
- Η μέθοδος A1 φαίνεται γενικά να είναι καλύτερη, καθώς είναι αρκετα λιγότερο περίπλοκη υπολογιστικά, δεν απαιτεί καποιο boundary condition και έχει παρόμοια η καλύτερα αποτελέσματα από τις υπόλοιπες, λογικό λοιπόν που ειναι αυτη που χρησιμοποιείται συχνότερα.

Κώδικας για 3.1import numpy as np

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from scipy import integrate
#Define the criteria given in the paper
def criterion_C1(pred, target, a=0, b=2):
    """
    Implements the C1 loss function.
    """
    z = pred
    phi = (b - a) / (1 + torch.exp(z)) + b * torch.log(1 +
torch.exp(z))
    psi = -torch.log(1 + torch.exp(z))
    return torch.mean(phi + target * psi)

def criterion_A1(pred, target):
    """
    Implements the A1 loss function.
    """
    return torch.mean(0.5 * pred**2 - target * pred)

class ShallowNetwork(nn.Module):
    def __init__(self, hidden_size=100):
        super().__init__()
        self.network = nn.Sequential(
            nn.Linear(1, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, 1)
        )

    def forward(self, x):
        return self.network(x)
#We define the reward function
def reward_function(s):
    return np.minimum(2, s**2)
```

```
#Calculate the trans densityu
def transition_density(Next_S, s, action):

    if action == 1:
        mean = 0.8 * s + 1.0
    else:
        mean = -2.0

    sigma = 1.0
    return (1 / (sigma * np.sqrt(2 * np.pi))) * np.exp(-0.5 *
((Next_S - mean) / sigma)**2)

## We make 1000 random actions then sort aproprietly for training
def generate_training_data(N=1000):

    S = np.zeros(N + 1)
    Next_S = np.zeros(N)
    actions = np.random.choice([1, 2], size=N)


    S[0] = np.random.standard_normal()

    for t in range(N):
        if actions[t] == 1:
            Next_S[t] = 0.8 * S[t] + 1.0 +
np.random.standard_normal()
        else:
            Next_S[t] = -2.0 + np.random.standard_normal()

        if t < N:
            S[t + 1] = Next_S[t]

    mask_a1 = actions == 1
    maska2 = actions == 2

    data_a1 = {'states': S[:-1][mask_a1],'next_states':
Next_S[mask_a1] }
```

```
    data_a2 = {'states': S[:-1][maska2],'next_states': Next_S[maska2]
}

    return data_a1, data_a2

#We define the numerical solution
## We use the built in nump method now for faster and more accurate
computeation
def numerical_solution(grid_points=1000):
    """
    Compute v₁(s) and v₂(s) using trapezoidal integration
    """
    s_grid = np.linspace(-20, 20, grid_points)
    ds = s_grid[1] - s_grid[0]
    t = 5
    v1 = np.zeros(grid_points)
    v2 = np.zeros(grid_points)

    for i, s in enumerate(s_grid):
        integral1 = lambda Next_S: reward_function(Next_S) *
transition_density(Next_S, s, 1)
        integrand2 = lambda Next_S: reward_function(Next_S) *
transition_density(Next_S, s, 2)

        v1[i] = np.trapz(integral1(s_grid), s_grid)
        v2[i] = np.trapz(integrand2(s_grid), s_grid)

    return s_grid, v1, v2

##trainj the nets we created
def train_networks(data_a1, data_a2, hidden_size=100, epochs=2000,
criterion_type="C1"):
    """
    Train two ShallowNetwork models (one per action) using either C1
or A1.
    """
    states1 = torch.FloatTensor(data_a1['states'].reshape(-1, 1))
    states2 = torch.FloatTensor(data_a2['states'].reshape(-1, 1))
```

```python
    rewards1 =
torch.FloatTensor(reward_function(data_a1['next_states']).reshape(-1,
1))
    rewards2 =
torch.FloatTensor(reward_function(data_a2['next_states']).reshape(-1,
1))

    net1 = ShallowNetwork(hidden_size)
    net2 = ShallowNetwork(hidden_size)

    optimizer1 = optim.Adam(net1.parameters(), lr=0.001)
    optimizer2 = optim.Adam(net2.parameters(), lr=0.001)

    criterion = criterion_C1 if criterion_type == "C1" else
criterion_A1

    losses1 = []
    losses2 = []
    #print(len(states1))
    #print(len(rewards1))
    for epoch in range(epochs):
        optimizer1.zero_grad()
        pred1 = net1(states1)
        loss1 = criterion(pred1, rewards1)
        loss1.backward()
        optimizer1.step()

        optimizer2.zero_grad()
        pred2 = net2(states2)
        loss2 = criterion(pred2, rewards2)
        loss2.backward()
        optimizer2.step()
        if epoch % 100 == 0:

            losses1.append(loss1.item())
            losses2.append(loss2.item())

            print(f'Epoch {epoch}/{epochs} ({criterion_type})')
            print(f'Action 1 - Loss: {loss1.item():.4f}')
```

```
            print(f'Action 2 - Loss: {loss2.item():.4f}')
            print('-' * 50)
    return net1, net2, (losses1, losses2)

def plot_loss(losses_C1, losses_A1):
    """
    Plot training progress (losses) for both C1 and A1 criteria.
    """
    plt.figure(figsize=(10, 5))
    ##########################################################
    plt.subplot(1, 2, 1)
    plt.plot(losses_C1[0], 'r-', label='Action 1')
    plt.plot(losses_C1[1], 'b-', label='Action 2')
    plt.title('C1 Optimization Losses')
    plt.xlabel('Epochs (×100)')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    #########################################################
    plt.subplot(1, 2, 2)
    plt.plot(losses_A1[0], 'r-', label='Action 1')
    plt.plot(losses_A1[1], 'b-', label='Action 2')
    plt.title('A1 Optimization Losses')
    plt.xlabel('Epochs (×100)')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.show()
## Ploting
def plot_comparison(s_grid, v1, v2, C1_a1_neural_net,
C1_a2_neural_net, a1_a1_neural_net, a1_a2_neural_net):
    s_tensor = torch.FloatTensor(s_grid.reshape(-1, 1))

    with torch.no_grad():
        # C1
        v1_nn_C1 = 2 *
torch.sigmoid(C1_a1_neural_net(s_tensor)).numpy()
```

```
        v2_nn_C1 = 2 *
torch.sigmoid(C1_a2_neural_net(s_tensor)).numpy()

        # A1
        v1_nn_A1 = a1_a1_neural_net(s_tensor).numpy()
        v2_nn_A1 = a1_a2_neural_net(s_tensor).numpy()

    plt.figure(figsize=(15, 10))

    plt.subplot(2, 2, 1)
    plt.plot(s_grid, v1, 'r-', label='v₁(s) Numerical')
    plt.plot(s_grid, v1_nn_C1, 'r--', label='v₁(s) C1')
    plt.plot(s_grid, v2, 'b-', label='v₂(s) Numerical')
    plt.plot(s_grid, v2_nn_C1, 'b--', label='v₂(s) C1')
    plt.title('Value Functions (C1)')
    plt.xlabel('State s')
    plt.ylabel('Value')
    plt.legend()
    plt.grid(True)
    plt.xlim(-5, 5)

    plt.subplot(2, 2, 2)
    plt.plot(s_grid, v1, 'r-', label='v₁(s) Numerical')
    plt.plot(s_grid, v1_nn_A1, 'r--', label='v₁(s) A1')
    plt.plot(s_grid, v2, 'b-', label='v₂(s) Numerical')
    plt.plot(s_grid, v2_nn_A1, 'b--', label='v₂(s) A1')

    plt.title('Value Functions (A1)')
    plt.xlabel('State s')
    plt.ylabel('Value')
    plt.legend()




    plt.grid(True)
    plt.xlim(-5, 5)

    plt.subplot(2, 2, 3)
```

```python
    optimal_action_num = np.where(v1 >= v2, 1, 2)
    optimal_action_nn_C1 = np.where(v1_nn_C1 >= v2_nn_C1, 1, 2)
    plt.plot(s_grid, optimal_action_num, 'r-', label='Numerical')
    plt.plot(s_grid, optimal_action_nn_C1, 'b--', label='C1')
    plt.title('Optimal Action Policy (C1)')
    plt.xlabel('State s')
    plt.ylabel('Action')
    plt.yticks([1, 2])
    plt.legend()
    plt.grid(True)
    plt.xlim(-5, 5)
    # --- Plot A1 policy ---
    plt.subplot(2, 2, 4)

    optimal_action_nn_A1 = np.where(v1_nn_A1 >= v2_nn_A1, 1, 2)
    plt.plot(s_grid, optimal_action_num, 'r-', label='Numerical')
    plt.plot(s_grid, optimal_action_nn_A1, 'b--', label='A1')

    plt.title('Optimal Action Policy (A1)')
    plt.xlabel('State s')
    plt.ylabel('Action')

    plt.yticks([1, 2])
    plt.legend()

    plt.grid(True)
    plt.xlim(-5, 5)

    plt.tight_layout()
    plt.show()
def plot_reward_function(reward_function, s_min=-5, s_max=5,
n_points=100):
    """
    Plot the reward function R(s) over [s_min, s_max].
    """
    s = np.linspace(s_min, s_max, n_points)
    r = reward_function(s)
    plt.figure(figsize=(6, 4))
```

```python
    plt.plot(s, r, 'r-', label='R(s)')

    plt.xlabel('State s')
    plt.ylabel('Reward')

    plt.title('Reward Function')

    plt.legend()
    plt.grid(True)
    plt.show()

if __name__ == "__main__":
    np.random.seed(42)
    torch.manual_seed(42)

    print("Generating training data...")
    data_a1, data_a2 = generate_training_data(N=1000)

    print("Computing numerical solution...")
    s_grid, v1, v2 = numerical_solution()

    print("Training networks with C1 criterion...")
    C1_a1_neural_net, C1_a2_neural_net, losses_C1 = train_networks(
        data_a1, data_a2, criterion_type="C1"
    )

    print("\nTraining networks with A1 criterion...")
    a1_a1_neural_net, a1_a2_neural_net, losses_A1 = train_networks(
        data_a1, data_a2, criterion_type="A1"
    )

    print("\nPlotting training progress...")
    plot_loss(losses_C1, losses_A1)

    print("Plotting comparison...")
    plot_comparison(s_grid, v1, v2, C1_a1_neural_net,
C1_a2_neural_net, a1_a1_neural_net, a1_a2_neural_net)

    plot_reward_function(reward_function)
```

Κώδικας για 2

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from scipy import integrate
# Loss Functions
def criterion_C1(pred, target, a=0, b=2):
    """
    Implements the C1 loss function.
    """
    z = pred
    phi = (b - a) / (1 + torch.exp(z)) + b * torch.log(1 +
torch.exp(z))
    psi = -torch.log(1 + torch.exp(z))
    return torch.mean(phi + target * psi)

def criterion_A1(pred, target):
    """
    Implements the A1 loss function.
    """
    return torch.mean(0.5 * pred**2 - target * pred)

class ShallowNetwork(nn.Module):
    """
    A simple neural network with one hidden layer.
    """
```

```python
    def __init__(self, hidden_size=100):
        super().__init__()
        self.network = nn.Sequential(
            nn.Linear(1, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, 1)
        )

    def forward(self, x):
        return self.network(x)

#Rewrard Function
def reward_function(s):

    return np.minimum(2, s**2)
#Transition Density
def transition_density(s_next, s_val, act):

    if act == 1:
        mean = 0.8 * s_val + 1.0
    else:
        mean = -2.0

    sigma = 1.0
    return (1 / (sigma * np.sqrt(2 * np.pi))) * np.exp(
        -0.5 * ((s_next - mean) / sigma) ** 2
    )

# Data Generation
def generate_training_data(N=1000):
    st = np.zeros(N + 1)
    st_next = np.zeros(N)
    acts = np.random.choice([1, 2], size=N)

    st[0] = np.random.standard_normal()

    for t in range(N):
        if acts[t] == 1:
            st_next[t] = 0.8 * st[t] + 1.0 +
```

```
np.random.standard_normal()
        else:
            st_next[t] = -2.0 + np.random.standard_normal()

        if t < N:
            st[t + 1] = st_next[t]

    mask1 = acts == 1
    mask2 = acts == 2

    data_a1 = {
        'states': st[:-1][mask1],
        'next_states': st_next[mask1]
    }
    data_a2 = {
        'states': st[:-1][mask2],
        'next_states': st_next[mask2]
    }

    return data_a1, data_a2

# Numerical Solution
def numerical_solution(grid_points=1000):
    svals = np.linspace(-20, 20, grid_points)
    ds = svals[1] - svals[0]

    v1_arr = np.zeros(grid_points)
    v2_arr = np.zeros(grid_points)

    for i, sv in enumerate(svals):
        integrand1 = lambda sn: reward_function(sn) *
transition_density(sn, sv, 1)
        integrand2 = lambda sn: reward_function(sn) *
transition_density(sn, sv, 2)

        v1_arr[i] = np.trapz(integrand1(svals), svals)
        v2_arr[i] = np.trapz(integrand2(svals), svals)

    return svals, v1_arr, v2_arr
```

```python
# Training
def train_networks(data_a1, data_a2, hidden_size=100, epochs=2000,
criterion_type="C1"):

    states1 = torch.FloatTensor(data_a1['states'].reshape(-1, 1))
    states2 = torch.FloatTensor(data_a2['states'].reshape(-1, 1))

    rewards1 =
torch.FloatTensor(reward_function(data_a1['next_states']).reshape(-1,
1))
    rewards2 =
torch.FloatTensor(reward_function(data_a2['next_states']).reshape(-1,
1))

    net1 = ShallowNetwork(hidden_size)
    net2 = ShallowNetwork(hidden_size)

    optim1 = optim.Adam(net1.parameters(), lr=0.001)
    optim2 = optim.Adam(net2.parameters(), lr=0.001)

    loss_func = criterion_C1 if criterion_type == "C1" else
criterion_A1

    loss_list1, loss_list2 = [], []

    for e in range(epochs):
        # Action 1  training
        optim1.zero_grad()
        p1 = net1(states1)
        l1 = loss_func(p1, rewards1)
        l1.backward()
        optim1.step()

        # Action 2  training
        optim2.zero_grad()
        p2 = net2(states2)
        l2 = loss_func(p2, rewards2)
        l2.backward()
```

```python
        optim2.step()

        # Log  every 100 epochs
        if e % 100 == 0:
            loss_list1.append(l1.item())
            loss_list2.append(l2.item())
            print(f'Epoch {e}/{epochs} ({criterion_type})')
            print(f'Action 1 - Loss: {l1.item():.4f}')
            print(f'Action 2 - Loss: {l2.item():.4f}')
            print('-' * 50)

    return net1, net2, (loss_list1, loss_list2)

def plot_loss(losses_C1, losses_A1):

    plt.figure(figsize=(10, 5))

    # C1 losses
    plt.subplot(1, 2, 1)
    plt.plot(losses_C1[0], 'r-', label='Action 1')
    plt.plot(losses_C1[1], 'b-', label='Action 2')
    plt.title('C1 Optimization Losses')
    plt.xlabel('Epochs (×100)')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)

    # A1 losses
    plt.subplot(1, 2, 2)
    plt.plot(losses_A1[0], 'r-', label='Action 1')
    plt.plot(losses_A1[1], 'b-', label='Action 2')
    plt.title('A1 Optimization Losses')
    plt.xlabel('Epochs (×100)')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.show()
```

```
def plot_comparison(svals, v1_arr, v2_arr, net1_C1, net2_C1, net1_A1,
net2_A1):

    s_tensor = torch.FloatTensor(svals.reshape(-1, 1))

    with torch.no_grad():
        # C1
        v1_c1 = 2 * torch.sigmoid(net1_C1(s_tensor)).numpy()
        v2_c1 = 2 * torch.sigmoid(net2_C1(s_tensor)).numpy()

        # A1
        v1_a1 = net1_A1(s_tensor).numpy()
        v2_a1 = net2_A1(s_tensor).numpy()

    plt.figure(figsize=(15, 10))

#####
    plt.subplot(2, 2, 1)
    plt.plot(svals, v1_arr, 'r-', label='v₁(s) Numerical')
    plt.plot(svals, v1_c1, 'r--', label='v₁(s) C1')
    plt.plot(svals, v2_arr, 'b-', label='v₂(s) Numerical')
    plt.plot(svals, v2_c1, 'b--', label='v₂(s) C1')
    plt.title('Value Functions (C1)')
    plt.xlabel('State s')
    plt.ylabel('Value')
    plt.legend()
    plt.grid(True)
    plt.xlim(-5, 5)
    #####
    plt.subplot(2, 2, 2)
    plt.plot(svals, v1_arr, 'r-', label='v₁(s) Numerical')
    plt.plot(svals, v1_a1, 'r--', label='v₁(s) A1')
    plt.plot(svals, v2_arr, 'b-', label='v₂(s) Numerical')
    plt.plot(svals, v2_a1, 'b--', label='v₂(s) A1')
    plt.title('Value Functions (A1)')
    plt.xlabel('State s')
    plt.ylabel('Value')
    plt.legend()
```

```python
    plt.grid(True)
    plt.xlim(-5, 5)
    ##########
    plt.subplot(2, 2, 3)
    opt_act_num = np.where(v1_arr >= v2_arr, 1, 2)
    opt_act_c1 = np.where(v1_c1 >= v2_c1, 1, 2)
    plt.plot(svals, opt_act_num, 'r-', label='Numerical')
    plt.plot(svals, opt_act_c1, 'b--', label='C1')
    plt.title('Optimal Action Policy (C1)')
    plt.xlabel('State s')
    plt.ylabel('Action')
    plt.yticks([1, 2])
    plt.legend()
    plt.grid(True)
    plt.xlim(-5, 5)
    ##################
    plt.subplot(2, 2, 4)
    opt_act_a1 = np.where(v1_a1 >= v2_a1, 1, 2)
    plt.plot(svals, opt_act_num, 'r-', label='Numerical')
    plt.plot(svals, opt_act_a1, 'b--', label='A1')
    plt.title('Optimal Action Policy (A1)')
    plt.xlabel('State s')
    plt.ylabel('Action')
    plt.yticks([1, 2])
    plt.legend()
    plt.grid(True)
    plt.xlim(-5, 5)

    plt.tight_layout()
    plt.show()

def plot_reward_function(rew_func, s_min=-5, s_max=5, n_points=100):

    svals = np.linspace(s_min, s_max, n_points)
    rewards = rew_func(svals)

    plt.figure(figsize=(6, 4))
    plt.plot(svals, rewards, 'r-', label='R(s)')
    plt.xlabel('State s')
```

```
    plt.ylabel('Reward')
    plt.title('Reward Function')
    plt.legend()
    plt.grid(True)
    plt.show()

if __name__ == "__main__":
    np.random.seed(42)
    torch.manual_seed(42)

    data1, data2 = generate_training_data(N=1000)

    svals, v1_arr, v2_arr = numerical_solution()

    net1_C1, net2_C1, losses_C1 = train_networks(data1, data2,
criterion_type="C1")

    net1_A1, net2_A1, losses_A1 = train_networks(data1, data2,
criterion_type="A1")

    plot_loss(losses_C1, losses_A1)

    plot_comparison(svals, v1_arr, v2_arr, net1_C1, net2_C1, net1_A1,
net2_A1)

    plot_reward_function(reward_function)
```

Code για 3.3

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from scipy import integrate
```

```python
class ShallowNetwork(nn.Module):
    def __init__(self, hidden_size=100):
        super().__init__()
        self.network = nn.Sequential(
            nn.Linear(1, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, 1)
        )

    def forward(self, x):
        return self.network(x)



def reward_function(s):
    """R(S) = min{2, S²}"""
    return np.minimum(2, s**2)


def transition_density(s_next, s, action):
    """p(s_next|s,α) - Gaussian transition density"""
    if action == 1:
        mean = 0.8 * s + 1.0
    else:
        mean = -2.0

    return (1/np.sqrt(2*np.pi)) * np.exp(-0.5 * (s_next - mean)**2)


def generate_training_data(N=1000):
    S = np.zeros(N+1)
    S_next = np.zeros(N)
    acts = np.random.choice([1, 2], size=N)

    S[0] = np.random.normal(0, 1)

    for t in range(N):
```

```
        if acts[t] == 1:
            S_next[t] = 0.8 * S[t] + 1.0 +
np.random.standard_normal()
        else:
            S_next[t] = -2.0 + np.random.standard_normal()

        if t < N:
            S[t+1] = S_next[t]

    mask1 = acts == 1
    mask2 = acts == 2

    data_a1 = {
        'states': S[:-1][mask1],
        'next_states': S_next[mask1]
    }

    data_a2 = {
        'states': S[:-1][mask2],
        'next_states': S_next[mask2]
    }

    return data_a1, data_a2



def numerical_solution(grid_points=1000, gamma=0.8, max_iter=1000,
tol=1e-6):
        K = 2
    s_grid = np.linspace(-8, 8, grid_points)
    ds = s_grid[1] - s_grid[0]

    V = np.zeros((grid_points, K))


    R = np.zeros((grid_points, K))
    for j in range(K):
        R[:, j] = reward_function(s_grid)
```

```
    s_next = np.linspace(-20, 20, grid_points * 2)
    ds_integration = s_next[1] - s_next[0]

    for iteration in range(max_iter):
        V_old = V.copy()

        F = np.zeros((grid_points, K))
        for j in range(K):
            for i, s in enumerate(s_grid):
                v_interp = np.interp(s_next, s_grid, np.max(V_old,
axis=1),
                                   left=np.max(V_old[0]),
right=np.max(V_old[-1]))

                integl = reward_function(s_next) *
transition_density(s_next, s, j+1) + \
                           gamma * v_interp *
transition_density(s_next, s, j+1)

                F[i, j] = np.trapz(integl, s_next)

        V = F

        rel_diff = np.max(np.abs((V - V_old) / (np.abs(V_old) +
1e-10)))
        if rel_diff < tol:
            print(f"Converged after {iteration+1} iterations")
            break

        if iteration % 100 == 0:
            print(f"Iteration {iteration}, relative diff:
{rel_diff:.6f}")

    return s_grid, V[:, 0], V[:, 1]
def criterion_C1(pred, target, a=0, b=20):
        z = pred
        phi = (b - a)/(1 + torch.exp(z)) + b * torch.log(1 +
torch.exp(z))
        psi = -torch.log(1 + torch.exp(z))
```

```
        c1_loss= torch.mean(phi + target * psi)

        monitoring_loss = torch.mean((phi - target)**2)
return c1_loss, monitoring_loss

def criterion_A1(pred, target):
                loss = torch.mean(0.5 * pred**2 - target * pred)


        main_loss = loss


        monitoring_loss = torch.mean((pred - target)**2)
        return main_loss, monitoring_loss
def train_networks_infinite(data_a1, data_a2, hidden_size=100,
epochs=2000, gamma=0.8, criterion="C1"):
    states1 = torch.FloatTensor(data_a1['states'].reshape(-1, 1))
    states2 = torch.FloatTensor(data_a2['states'].reshape(-1, 1))
    next_states1 =
torch.FloatTensor(data_a1['next_states'].reshape(-1, 1))
    next_states2 =
torch.FloatTensor(data_a2['next_states'].reshape(-1, 1))





    if criterion == "C1":
        net1 = ShallowNetwork(hidden_size)
        net2 = ShallowNetwork(hidden_size)

        optimizer1 = optim.Adam(net1.parameters(), lr=0.0001)
        optimizer2 = optim.Adam(net2.parameters(), lr=0.0001)

        scheduler1 =
torch.optim.lr_scheduler.CosineAnnealingLR(optimizer1, epochs,
eta_min=1e-6)
        scheduler2 =
torch.optim.lr_scheduler.CosineAnnealingLR(optimizer2, epochs,
```

```
eta_min=1e-6)

        def train_step(net, optimizer, states, next_states, rewards):
            with torch.no_grad():
                v1_next = 20 * torch.sigmoid(net1(next_states))
                v2_next = 20 * torch.sigmoid(net2(next_states))
                next_values = torch.maximum(v1_next, v2_next)


            target = rewards + gamma * next_values

            optimizer.zero_grad()
            pred = net(states)
            loss, monitoring_loss = criterion_C1(pred, target)

            if torch.isfinite(loss):
                loss.backward()


                optimizer.step()

            return loss, monitoring_loss

    else:
        net1 = ShallowNetwork(hidden_size)
        net2 = ShallowNetwork(hidden_size)
        optimizer1 = optim.Adam(net1.parameters(), lr=0.0005,
weight_decay=0.001)
        optimizer2 = optim.Adam(net2.parameters(), lr=0.0005,
weight_decay=0.001)

        def train_step(net, optimizer, states, next_states, rewards):
            v1_next = net1(next_states)
            v2_next = net2(next_states)
            next_values = torch.maximum(v1_next.detach(),
v2_next.detach())

            target = rewards + gamma * next_values
            optimizer.zero_grad()
```

```
            pred = net(states)
            loss, monitoring_loss = criterion_A1(pred, target)
            loss.backward()

            optimizer.step()
            return loss, monitoring_loss


    l1, l2 = [], []
    monitoring_l1, monitoring_l2 = [], []



    for epoch in range(epochs):
        rewards1 =
torch.FloatTensor(reward_function(data_a1['next_states']).reshape(-1,
1))
        rewards2 =
torch.FloatTensor(reward_function(data_a2['next_states']).reshape(-1,
1))


        loss1, mon_loss1 = train_step(net1, optimizer1, states1,
next_states1, rewards1)
        loss2, mon_loss2 = train_step(net2, optimizer2, states2,
next_states2, rewards2)


        if criterion == "C1":
            scheduler1.step()
            scheduler2.step()

        if epoch % 10 == 0:
            l1.append(loss1.item())
            l2.append(loss2.item())
            monitoring_l1.append(mon_loss1.item())
            monitoring_l2.append(mon_loss2.item())
```

```python
    return net1, net2, (l1, l2), (monitoring_l1, monitoring_l2)
def plot_training_progress(losses_orig, monitoring_losses_orig,
losses_A1, monitoring_losses_A1):
    """Plot training progress for both criteria"""
    plt.figure(figsize=(15, 10))

    # C1 criterion plots
    plt.subplot(2, 2, 1)
    plt.plot(losses_orig[0], 'r-', label='Action 1')
plt.plot(losses_orig[1], 'b-', label='Action 2')
    plt.title('C1 Optimization Losses')
    plt.xlabel('Epochs (×100)')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)

    plt.subplot(2, 2, 2)
    plt.plot(monitoring_losses_orig[0], 'r-', label='Action 1')
    plt.plot(monitoring_losses_orig[1], 'b-', label='Action 2')
    plt.title('C1 Mean Squared Error')
    plt.xlabel('Epochs (×100)')
    plt.ylabel('MSE')
    plt.legend()
    plt.grid(True)

    # A1 criterion plots
    plt.subplot(2, 2, 3)
    plt.plot(losses_A1[0], 'r-', label='Action 1')
    plt.plot(losses_A1[1], 'b-', label='Action 2')
    plt.title('A1 Optimization Losses')
    plt.xlabel('Epochs (×100)')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)

    plt.subplot(2, 2, 4)
    plt.plot(monitoring_losses_A1[0], label='Action 1', )
    plt.plot(monitoring_losses_A1[1], 'b-', label='Action 2')
```

```python
    plt.title('A1 Mean Squared Error')
    plt.xlabel('Epochs (×100)')
    plt.ylabel('MSE')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.show()




def plot_comparison(s_grid, v1, v2, net1_orig, net2_orig, net1_A1,
net2_A1):
    """Plot results comparing both criteria"""
    s_tensor = torch.FloatTensor(s_grid.reshape(-1, 1))

    with torch.no_grad():
        # C1 predictions
        c1out = 0*1/(1+torch.exp(net1_orig(s_tensor))) +
20*torch.exp(net1_orig(s_tensor)) /
(1+torch.exp(net1_orig(s_tensor)))
        c2out = 0*1/(1+torch.exp(net2_orig(s_tensor))) +
20*torch.exp(net2_orig(s_tensor)) /
(1+torch.exp(net2_orig(s_tensor)))
        v1_nn_orig = c1out
        v2_nn_orig = c2out

        # A1 predictions
        v1_nn_A1 = net1_A1(s_tensor).numpy()
        v2_nn_A1 = net2_A1(s_tensor).numpy()

    plt.figure(figsize=(15, 10))

    # Plot C1 results
    plt.subplot(2, 2, 1)
    plt.plot(s_grid, v1, 'r-', label='v₁(s) Numerical')
    plt.plot(s_grid, v1_nn_orig, 'r--', label='v₁(s) C1')
    plt.plot(s_grid, v2, 'b-', label='v₂(s) Numerical')
```

```
    plt.plot(s_grid, v2_nn_orig, 'b--', label='v₂(s) C1')
    plt.title('Value Functions (C1)')
    plt.xlabel('State s')
    plt.ylabel('Value')
    plt.legend()
    plt.grid(True)
    plt.xlim(-5, 5)

    # Plot A1 results
    plt.subplot(2, 2, 2)
    plt.plot(s_grid, v1, 'r-', label='v₁(s) Numerical')
    plt.plot(s_grid, v1_nn_A1, 'r--', label='v₁(s) A1')
    plt.plot(s_grid, v2, 'b-', label='v₂(s) Numerical')
    plt.plot(s_grid, v2_nn_A1, 'b--', label='v₂(s) A1')
    plt.title('Value Functions (A1)')
    plt.xlabel('State s')
    plt.ylabel('Value')
    plt.legend()
    plt.grid(True)
    plt.xlim(-5, 5)

    # Plot C1 policy
    plt.subplot(2, 2, 3)
    optimal_action_num = np.where(v1 >= v2, 1, 2)
    optimal_action_nn_orig = np.where(v1_nn_orig >= v2_nn_orig, 1, 2)

    plt.plot(s_grid, optimal_action_num, 'r-', label='Numerical')
    plt.plot(s_grid, optimal_action_nn_orig, 'b--', label='C1')
    plt.title('Optimal Action Policy (C1)')
    plt.xlabel('State s')
    plt.ylabel('Action')
    plt.yticks([1, 2])
    plt.legend()
    plt.grid(True)
    plt.xlim(-5, 5)


    plt.subplot(2, 2, 4)
    optimal_action_nn_A1 = np.where(v1_nn_A1 >= v2_nn_A1, 1, 2)
```

```
    plt.plot(s_grid, optimal_action_num, 'r-', label='Numerical')
    plt.plot(s_grid, optimal_action_nn_A1, 'b--', label='A1')
    plt.title('Optimal Action Policy (A1)')
    plt.xlabel('State s')
    plt.ylabel('Action')
    plt.yticks([1, 2])
    plt.legend()
    plt.grid(True)
    plt.xlim(-5, 5)

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
        np.random.seed(42)
    torch.manual_seed(42)


    data_a1, data_a2 = generate_training_data(N=1000)
      s_grid, v1, v2 = numerical_solution()

       net1_orig, net2_orig, losses_orig, monitoring_losses_orig =
train_networks_infinite(
        data_a1, data_a2, criterion="C1"
    )

        net1_A1, net2_A1, losses_A1, monitoring_losses_A1 =
train_networks_infinite(
        data_a1, data_a2, criterion="A1"
    )

        plot_training_progress(losses_orig, monitoring_losses_orig,
losses_A1, monitoring_losses_A1)


    plot_comparison(s_grid, v1, v2, net1_orig, net2_orig, net1_A1,
net2_A1)
```