ERCİYES ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ A.B.D.
BİM-627 DERİN ÖĞRENME-I DERSİ
ARAŞTIRMA ÖDEVİ

“REINFORCEMENT LEARNING PRATİK UYGULAMA”

SNAKE GAME

Ders Koordinatörü: Prof. Dr. Alper BAŞTÜRK

Dersi Alan Öğrenci: Ahmet Utku ELİK, 4010940078

1. SNAKE GAME

1.1. Tanım

Snake oyunu, genellikle 2D bir ızgara üzerinde oynanan ve yılanın yemleri yemesiyle büyüdüğü, ancak kendi vücuduna veya çevreye çarptığında oyunun sona erdiği basit ama etkileyici bir bilgisayar oyunudur. Oyuncu, yılanı dört farklı yönde hareket ettirerek yemleri toplamaya çalışır. Oyun süresince yılanın boyu uzadıkça hareket alanı azalır, bu da oyunu giderek zorlaştırmaktadır.

1.2. Oyun Mekanizması

Snake oyununun temel amacı, yılanın başını ekrandaki yemlere yönlendirmek ve yemleri yedikçe puan toplamaktır. Oyuncu, yılanın yönünü kontrol ederek aşağıdaki hedeflere ulaşmaya çalışır:

- **Yem Yeme:** Her yem yendiğinde yılanın boyu uzar ve puan artar.
- **Çarpışmadan Kaçınma:** Yılanın kendi vücuduna veya oyun sınırlarına çarpması oyunu sonlandırır.

Snake oyunu, genellikle bir ızgara (grid) sistemi üzerinde oynanan ve yılanın yemleri toplayarak büyüdüğü, ancak kendi vücuduna veya oyun sınırlarına çarptığında sona eren bir oyun yapısına sahiptir. Oyun alanı, belirli bir genişlik ve yükseklikte bir dikdörtgen alan olarak tanımlanır. Bu alan, genellikle her biri sabit boyutlarda (örneğin, 20x20 piksel) hücrelere bölünmüştür. Yılan başlangıçta bu ızgara sisteminin merkezine yakın bir yerde belirir ve belirli bir uzunlukta, bir baş ve birkaç gövde segmentinden oluşur. Yem ise yılanın yemesi gereken hedef olarak rastgele bir hücreye yerleştirilir. Yılan her yem yediğinde, yem başka bir rastgele konuma taşınır ve yılan bir segment uzunluğunda büyür.

Oyun sırasında yılan, sürekli bir yönde hareket eder. Oyuncu, yılanın yönünü klavye tuşları (örneğin, ok tuşları veya WASD) ile kontrol edebilir. Ancak yılan ters yöne dönemez; bu, yılanın başının hemen arkasındaki segmente dönerek çarpışmasını önler. Yılanın hareketi sırasında çeşitli çarpışma kontrolleri yapılır. Yılanın başı, oyun alanının sınırlarına çarparsa veya kendi gövdesinin herhangi bir segmentiyle çakışırsa oyun sona erer. Çarpışma durumu dışında yılan, her hareket adımında yemlere ulaşmaya çalışır. Yılanın başı, yemle aynı konuma geldiğinde yem yenmiş sayılır, yılanın boyu uzar ve oyuncunun puanı artar.

Oyun zamanlaması, belirli bir hızda çalışacak şekilde ayarlanmıştır. Genellikle oyun hızı, saniyede işlenen kare (frame per second, FPS) sayısı ile ifade edilir. Örneğin, bir oyun saniyede 40 kare hızında çalışabilir. Bu hız, sabit tutulabilir ya da yılanın uzunluğu arttıkça oyun daha zor hale gelsin diye yavaş yavaş artırılabilir. Zamanlama, oyunun akıcılığını ve zorluk seviyesini kontrol etmek için kritik bir rol oynar.

Oyuncu, her oyun adımında yılanın yönünü değiştirebilir. Bu değişiklikler, oyunun stratejik yapısını oluşturur. Yılanın yemlere ulaşabilmesi ve kendisine çarpmadan uzun süre hayatta kalabilmesi için oyuncunun dikkatli bir şekilde hareket planlaması yapması gerekir. Bu mekanik, oyunu hem basit hem de son derece bağımlılık yaratan bir hale getirir. Sonuç olarak, Snake oyununun bu temel çalışma prensipleri, oyunu hem eğlenceli hem de öğrenmesi kolay bir hale getirirken oynadıkça ustalaşmayı gerektiren bir yapıya dönüştürür.

Snake oyununun temel döngüsü şu şekildedir;

1. **Kullanıcı Girdisi:** Oyuncunun yön tuşlarıyla kontrol ettiği hareketler alınır.
2. **Yılanın Hareketi:** Yılanın başı yeni bir konuma taşınır.
3. **Çarpışma Kontrolü:** Yılanın sınırlarla veya kendi gövdesiyle çarpışıp çarpışmadığı kontrol edilir.
4. **Yem Yeme Kontrolü:** Yılanın başı yemeğin bulunduğu konuma ulaşıp ulaşmadığı kontrol edilir.
5. **Ekranın Güncellenmesi:** Yılanın yeni konumu ve yemin yeni konumu ekranda görüntülenir.
6. **Oyun Hızı:** Belirtilen FPS (Frame Per Second) hızına göre döngü bekletilir.

1.3. Temel Parametre ve Bileşenler

Oyun içerisindeki temel parametreler, oyun alanının genişliği ve yüksekliği olan **Ekran Boyutları**, yılanın her bir segmentinin ve yemin boyutu olan **Blok Boyutu** ve son olarak oyun döngüsünün çalıştırılma hızı (FPS) **Hız** değerleridir. Geliştirilen uygulama 640x480px Ekran Boyutuna, 20x20px Blok Boyutuna ve 40 fps Hız değerine sahiptir.

Snake oyunu içerisindeki temel bileşenler ise **yılan**, **yem** ve **puan** değeridir. Yılan bir liste olarak temsil edilir ve her segment Point(x, y) koordinatları ile tanımlanır. Yem, rastgele bir konumda oluşturulur ve yılanın yemesi beklenir. Puan değeri ise oyun içerisinde yılanın yediği yem adedini temsil eder.

1.4. Veri Yapısı ve Hareket Mekanizması

Yılan, uygulama içerisinde genellikle bir liste olarak temsil edilir. Listenin ilk elemanı yılanın başını, diğer elemanlar ise yılanın gövde segmentlerini temsil eder. Her eleman, yılanın bir segmentinin koordinatlarını belirtir. Örneğin

```
self.snake = [
    Point(100, 100),  # Baş
    Point(80, 100),   # İlk gövde segmenti
    Point(60, 100)    # Son gövde segmenti (Kuyruk)
]
```

Yılanın hareket mekanizması ise x ekseninde sol/sağ, y ekseninde ise yukarı/aşağı şeklinde her bir iterasyonda **Blok Boyutu** olan **20px**'lik adımlar ile gerçekleşmektedir. Hareket ve konum güncelleme işlemleri yılanın **başına**, **kuyruğuna** ve arada kalan **gövde segmentlerine** farklı algoritmik yapılar ile uygulanır.

Yılanın başı, seçilen hareket yönüne göre yeni bir pozisyona taşınır ve yeni baş pozisyonu, listenin başına eklenir. Örneğin yılan sağa hareket ediyor ve başın pozisyonu (100, 100) ise yeni baş pozisyonu (120, 100) olur ve listeye eklenir.

Yılanın son segmenti olan kuyruğu, hareket ve konum güncelleme sırasında yılanın yem yeme koşuluna göre güncellenir. Eğer hareket sırasında yılan yem yemez ise kuyruk listenin son elemanından çıkarılır böylelikle yılanın uzunluğu sabit kalır. Eğer yılan bir yem yerse, kuyruğun son segmenti çıkarılmaz ve yılan bir segment uzunluğunda (20px) büyür.

Örneğin yukarıda örnek olarak verilen yılan veri yapısına göre yılanın sağ tarafa 1 iterasyon yem yemeden ilerleme durumunda veri yapısı aşağıdaki gibi olur.

```
    self.snake = [
        Point(120, 100),  # Güncel baş değeri listeye eklendi!
        Point(100, 100),  # İlk gövde segmenti, önceki baş segmenti.
        Point(80, 100)    # Son gövde segmenti (Kuyruk)
      # Point(60, 100)    # Önceki kuyruk segmenti listeden çıkarıldı!
    ]
```

Eğer yılanın sağ tarafa hareketi esnasında yem yerse güncel baş değeri listeye eklenir ve diğer gövde segment değerleri segment uzunluğu kadar güncellenir;

```
    self.snake = [
        Point(120, 100),  # Güncel baş değeri listeye eklendi!
        Point(100, 100),  # İlk gövde segmenti, önceki baş segmenti.
        Point(80, 100)    # İkinci gövde segmenti.
        Point(60, 100)    # Son gövde segmenti (Kuyruk) yem yendiği için çıkarılmadı!
    ]
```

Son olarak yılanın gövde segmentleri, bir önceki segmentin önceki pozisyonunu takip eder. Her hareket adımında gövde segmentleri bir sıra halinde bir önceki segment değerine kaydırılır. Bu adım, başın hareket ettiği iz boyunca gövdenin hareket etmesini sağlar.

Yılanın oyun alanı içerisinde hareketi esnasında kendi gövdesine çarpması durumunda oyun sonlanır. Bunun için yılanın hareketi ve konumu güncellenirken gövde segmentlerine çarpma kontrolünün yapılması gerekmektedir. Yılanın kendi gövdesine çarpıp çarpmadığını kontrol etmek için, başın pozisyonu gövdenin diğer segmentleri ile karşılaştırılır:

```
    def is_collision(self, pt=None):
        if pt is None:
            pt = self.head
        if pt in self.snake[1:]:  # Başın gövde segmentleriyle çakışması
            return True
        return False
```

Her bir iterasyonda is_collision() metodu ile çarpışma kontrolü yapılır ve çarpışma durumunda oyun sonlandırılır.

2. REINFORCEMENT LEARNING - SNAKE GAME

2.1. Giriş

Pekiştirmeli Öğrenme (Reinforcement Learning, RL), bir ajanın çevresiyle etkileşime girerek deneyimlerden öğrendiği ve ödülleri maksimize etmeyi amaçladığı bir makine öğrenmesi yöntemidir. Deep Q-Networks (DQN) ise, RL'nin değer tabanlı bir varyantıdır ve derin öğrenme algoritmalarıyla birleştirilerek büyük ve karmaşık durum uzaylarını öğrenebilme yeteneğine sahiptir. Snake oyunu gibi sürekli değişen dinamik bir ortamda, DQN, optimal hareketleri öğrenmek için etkili bir araçtır. Bu projede, DQN kullanılarak Snake oyununda bir ajanın yemleri toplarken çarpışmalardan kaçınmayı öğrenmesi amaçlanmıştır.

2.2. Reinforcement Learning'in Snake Oyununa Entegrasyonu

Snake oyunu, pekiştirmeli öğrenme (Reinforcement Learning, RL) algoritmalarının test edilmesi ve geliştirilmesi için ideal bir ortam sunar. Bu oyunun basit kuralları ve karmaşık dinamikleri, RL algoritmalarının öğrenme yeteneklerini ve karar alma süreçlerini değerlendirmek için etkili bir platform sağlar. Snake oyunundaki temel hedef, yılanın rastgele konumlandırılmış yemleri toplaması ve her yem yendiğinde boyunun uzamasıdır. Ancak oyunun ilerleyen aşamalarında yılanın hareket alanı daralır ve oyun, daha stratejik kararlar gerektirir. Bu durum, RL algoritmalarının keşif ve sömürü dengesini öğrenmesini teşvik eder.

Reinforcement Learning'in Temel Unsurları

Snake oyununu RL algoritmalarına uygun bir çevre haline getiren temel bileşenler durum (state), aksiyon (action) ve ödül (reward) unsurlarıdır.

I. **Durum (State):** Durum, oyunun mevcut anındaki tüm bilgileri kapsar ve RL algoritmasına giriş verisi olarak kullanılır. Snake oyunu için durum, aşağıdaki öğeleri içerebilir:

- **Yılanın Pozisyonu:** Yılanın başı ve gövde segmentlerinin koordinatları.
- **Hareket Yönü:** Yılanın mevcut hareket yönü (sağa, sola, yukarı veya aşağı).
- **Yemin Konumu:** Yemin yılanın başına göre göreceli pozisyonu (solda, sağda, yukarıda veya aşağıda).
- **Çarpışma Riski:** Yılanın mevcut yönünde, sağında veya solunda çarpışma ihtimali.

Bu bilgiler, bir durum vektörü şeklinde RL modeline giriş olarak verilir. Örnek bir durum vektörü:

```
state = [0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0] # Durum (state) vektörü
```

Bu durum (state) vektöründe;

- İlk üç eleman, çarpışma risklerini temsil eder.
- Sonraki dört eleman, yılanın hareket yönünü temsil eder.
- Son dört eleman, yemin göreceli konumunu temsil eder.

II. **Aksiyon (Action):** Aksiyon, yılanın bir sonraki hareket yönünü ifade eder. Oyundaki olası aksiyonlar:

- **[1, 0, 0]:** İleri git (hareket yönünü değiştirme).
- **[0, 1, 0]:** Sağa dön (mevcut yönün bir sonraki saat yönüne doğru yönüne geçiş).
- **[0, 0, 1]:** Sola dön (mevcut yönün bir önceki saat yönüne doğru yönüne geçiş).

RL ajanının (agent) eğitimi boyunca aksiyonlar epsilon-greedy politikası ile belirlenmektedir. Bu politika ile rastgele bir aksiyon seçilmesi durumunda keşif (exploration) güçlenir ve ajan, rastgele aksiyonlar seçerek çevresini keşfeder. Ajanın eğitildiği oyun sayısı arttıkça epsilon değeri azalır ve sömürü (exploitation) güçlenir. Bu aşamada ajan, DQN modelinin tahmin ettiği en yüksek Q-değerine sahip aksiyonu seçer.

III. **Ödül (Reward):** Ödül, ajanın gerçekleştirdiği aksiyonun sonucunda aldığı geri bildirimdir ve RL algoritmasının öğrenmesini yönlendiren temel unsurdur. Snake oyunu için ödül mekanizması şu şekilde yapılandırılabilir:

- **Pozitif Ödül (+10):** Yılanın yemi yemesi durumunda verilir.
- **Negatif Ödül (-10):** Yılanın kendi vücuduna veya oyun sınırlarına çarpması durumunda verilir.
- **Küçük Negatif Ödül (-1):** Yılanın her hareket adımında, enerji kaybını simüle etmek ve optimal çözümleri teşvik etmek için uygulanabilir.

Ödül mekanizması ile ajanın uzun vadeli stratejik kararlar alması teşvik edilir.

Snake Oyununun RL Çevresi Haline Getirilmesi

Snake oyunu, yukarıdaki durum, aksiyon ve ödül bileşenlerinin bir araya getirilmesiyle RL algoritmalarının optimal politikalar öğrenmesini sağlayan bir çevre haline getirilir. Bu yapı, aşağıdaki unsurları içerir:

- **Çevre (Environment):** Oyun, yılanın hareket edebileceği ve yemleri toplayabileceği bir çevre olarak yapılandırılır.
- **Ajan (Agent):** Yılanın kontrolünü sağlayan RL algoritmasıdır. Ajan, durum-aksiyon-ödül döngüsünü kullanarak öğrenir.
- **Eğitim Süreci:** Ajan, oyunda deneyim kazandıkça, daha iyi kararlar almayı öğrenir.

2.3. Epsilon-Greedy Politikası

Epsilon-greedy politikası, pekiştirmeli öğrenme (Reinforcement Learning, RL) algoritmalarında, keşif (exploration) ve sömürü (exploitation) arasındaki dengeyi sağlamak için kullanılan bir aksiyon seçme stratejisidir. Bu strateji, bir ajanın, uzun vadeli ödülü maksimize etmek için çevresini keşfetmesini (yeni aksiyonları denemesini) ve mevcut bilgilerinden yararlanmasını (öğrendiği optimal aksiyonları uygulamasını) dengeler.

Bu politikada keşif ve sömürü yapıları aşağıdaki şekilde işlenir;

Keşif (Exploration):

- Ajan, mevcut bilgisine dayanarak en iyi aksiyonu seçmek yerine rastgele aksiyonlar seçer.
- Bu süreç, ajanın yeni durumları ve ödülleri keşfetmesine olanak tanır.
- Keşif, uzun vadeli öğrenmeyi ve daha geniş bir bilgi tabanını garanti eder.

Sömürü (Exploitation):

- Ajan, mevcut bilgiye dayanarak en yüksek ödülü sağlayacak aksiyonu seçer.
- Bu süreç, ajanın kısa vadede ödül kazanmasını maksimize eder.
- Ancak, keşif yapılmadığında ajan, optimal olmayan politikaları öğrenebilir.

Epsilon-greedy politikasında yer alan ε (epsilon) değeri rastgele aksiyon seçme olasılığını kontrol eden bir hiperparametredir. Bu değer genellikle 0-1 aralığında seçilir. Örneğin ε = 0.1 olduğunda ajan %10 olasılıkla rastgele bir aksiyon seçer, %90 olasılıkla ise mevcut bilgilerinden yararlanarak en iyi aksiyonu seçer. Bu işlemi matematiksel olarak aşağıdaki şekilde ifade edebiliriz;

$$a_t = \begin{cases} Random\ Action & if\ r < \varepsilon \\ \arg max_a Q(s_t, a) & if\ r \geq \varepsilon \end{cases}$$

- a_t : Zaman t'de seçilen aksiyon.
- $Q(s_t, a)$: Durum s_t'de aksiyon a'nın Q-değeri.
- ε (Epsilon) : Keşif olasılığı.

RL ajanına uygulanan Epsilon-Greedy politikasındaki epsilon (ε) değeri zamanla değişerek keşif ile sömürü arasındaki dengeyi sağlamaktadır. Epsilon değerinin azaltılması için yaygın olarak lineer, üstel ve adım azaltma yöntemleri kullanılmaktadır.

Geliştirilen uygulamada ajanın gerçekleştireceği aksiyon seçiminde Epsilon-Greedy politikası uygulanmıştır. Epsilon değeri ajanın eğitildiği mevcut oyun sayı değeri ile azaltılmaktadır. İlk 80 oyunda keşfi ön plana çıkarmak için 80 değerinden mevcut oyun değeri farkı alınmıştır.

```
self.epsilon = 80 - self.n_games          # Epsilon zamanla azalır.
if random.randint(0, 200) < self.epsilon:
     # Ajan, rastgele aksiyonlar seçerek çevresini keşfeder.
else:
     # Ajan, DQN modelinin tahmin ettiği en yüksek Q-değerine sahip aksiyonu seçer.
```

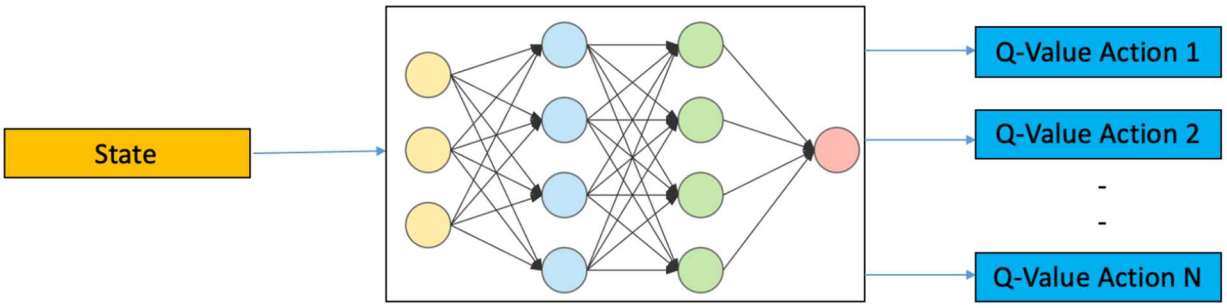
Bu yapıda yüksek epsilon değeri ile oyunun başlangıcında rastgele aksiyonlar seçilmesi sağlanır ve keşif kabiliyeti artırılır. Zamanla epsilon azalır, bu da ajanın sinir ağı modeline güvenerek daha optimal aksiyonlar seçmesini teşvik eder böylelikle ilerleyen eğitim oyunlarında sömürü kabiliyeti artırılmaktadır.

2.4.RL DQN Yönteminin Snake Oyununa Entegrasyonu

Deep Q-Networks (DQN), klasik Q-Learning algoritmasının derin sinir ağlarıyla birleşiminden oluşan, geniş ve karmaşık durum-aksiyon uzaylarını öğrenebilme kapasitesine sahip bir yöntemdir. Snake oyununda DQN kullanarak, yılanın optimal hareketleri öğrenmesi ve bu hareketleri uygulayarak oyun boyunca hayatta kalması sağlanır. DQN, temel olarak aşağıdaki adımlardan oluşur:

I. Durum-Aksiyon Değerlerinin Tahmini

DQN modelinde bir derin sinir ağı, her durum için olası aksiyonların Q-değerlerini tahmin eder. Q-değeri, belirli bir durum-aksiyon çiftinin beklenen ödülünü ifade eder. Snake oyunu için:



Şekil 1. DQN Model

- **Girdi:** Durum vektörü (örneğin, yılanın pozisyonu, yönü, yem konumu ve çarpışma riski gibi bilgileri içeren bir dizi, 11 features).

- **Çıktı:** Üç aksiyonun (ileri git, sağa dön, sola dön) Q-değerlerini içeren bir vektör.

```
Q_values = [3.5, 2.1, -1.2]  # İleri git, sağa dön, sola dön.
```

Yukarıda verilen örnek sinir ağı çıktısını (Q-değerleri) inceleyecek olursak ajan, eylem seçimini en yüksek Q-değerine sahip olan aksiyonu seçer yani Q_values[0] →ileri git şeklinde.

II. Hedef Q-Değerinin Hesaplanması

DQN algoritması, Bellman denklemini kullanarak hedef Q-değerlerini hesaplar. Bellman denklemi aşağıda verilmiştir;

$$\boldsymbol{Q_{new}} = r + \gamma \max(Q_{next})$$

- Q_{new} : Güncellenmiş hedef Q-değeri.
- r : Oyun adımında alınan ödül.
- γ : İndirim faktörü (gelecekteki ödüllerin bugünkü değerine etkisini kontrol eder).
- $\max(Q_{next})$: Sonraki durumda alınabilecek maksimum Q-değeri.

Örneğin alınan ödül (r) yem yenilmesi durumunda +10, indirim faktörü (γ) 0.9 ve sonraki durumdaki max Q-değeri 5.0 için hedef Q-değeri aşağıdaki şekilde hesaplanır;

$$\boldsymbol{Q_{new}} = 10 + 0.9\ x\ 5.0 = 14.5$$

III. Model Güncellenmesi

Tahmin edilen Q-değerleri (Q_{pred}) ile hedef Q-değerleri (Q_{new}) arasındaki fark, bir kayıp fonksiyonu ile hesaplanır. Bu fark, modelin öğrenme sürecini yönlendirir. Bu projede kayıp fonksiyonu olarak aşağıda denklem yapısı verilen MSE kullanılarak kayıp değerleri hesaplanmıştır. Denklemde yer alan N değeri eğitimde kullanılan örnek sayısı yani oyun değerini ifade eder.

$$\boldsymbol{Loss} = \frac{1}{N}\sum_{i=1}^{N}\left(Q_{new} - Q_{pred}\right)^2$$

IV. Deneyim Tekrarı

Deneyim Tekrarı (Experience Replay), pekiştirmeli öğrenme (Reinforcement Learning, RL) algoritmalarında, geçmişte yaşanan deneyimlerin bir bellek (replay buffer) içinde saklanması ve bu deneyimlerin modelin eğitimi sırasında rastgele bir şekilde tekrar kullanılmasıdır. Bu yöntem, eğitim kararlılığını artırır, veri korelasyonunu azaltır ve RL modellerinin öğrenme performansını iyileştirir. Snake oyununda bu yapı, yılanın çevresel durumlardan öğrenmesini daha etkili bir şekilde sağlar. Bu amaçla deneyim belleği kullanılmaktadır.

Deneyim Belleği

Deneyim Belleği, oyun sırasında toplanan durum, aksiyon, ödül, sonraki durum ve oyun durumu (done) bilgilerinin saklandığı bir FIFO (First-In-First-Out) veri yapısıdır. Belleğin amacı, geçmiş deneyimlerden öğrenmeyi mümkün kılarak daha geniş bir veri tabanı oluşturmak ve modelin genelleme yeteneğini artırmaktır. Bellek, genellikle sabit bir maksimum kapasiteye sahiptir (örneğin bbu projede, 100.000 deneyim). Yeni bir deneyim belleğe eklendiğinde, eğer kapasite doluysa en eski deneyim çıkarılır.

Bellekte Saklanan Bilgiler

Her deneyim sonucu bellekte saklanan bilgiler aşağıdaki bilgileri içerir.

- **Durum (state):** Oyunun mevcut durumu (örneğin, yılanın pozisyonu, yem konumu, çarpışma riski).
- **Aksiyon (action):** Yılanın seçtiği hareket (örneğin, ileri git, sağa dön).
- **Ödül (reward):** Aksiyon sonucunda alınan ödül (örneğin, +10 yem yendiğinde, -10 çarpışma olduğunda).
- **Sonraki Durum (next_state):** Aksiyon sonrası oyunun yeni durumu.
- **Oyun Durumu (done):** Oyunun bitip bitmediği bilgisi.

Rastgele Örnekleme

Deneyim tekrarında, eğitim sırasında bellekten rastgele bir mini-batch seçilir. Rastgele örnekleme:

- **Veri Korelasyonunu Azaltır:** Eğitim verilerindeki ardışıklığı ortadan kaldırır, bu da modelin daha genelleştirilmiş bir öğrenme yapmasını sağlar.

- **Daha Kararlı Öğrenme Sağlar:** Model, farklı durum-aksiyon çiftlerinden gelen verilerle eğitildiği için kararlı bir şekilde öğrenir.

Mevcut projede bu yapı aşağıdaki şekilde kullanılmıştır;

```
if len(self.memory) > BATCH_SIZE:
    mini_sample = random.sample(self.memory, BATCH_SIZE)
else:
    mini_sample = self.memory
```

Eğer bellekte yeterli sayıda deneyim varsa, rastgele bir mini-batch seçilir. Aksi halde mevcut tüm deneyimler kullanılır. Çalışmada MAX_MEMORY = 100_000, BATCH_SIZE = 1000 olarak kullanılmıştır.

Eğitim Süreci

Seçilen mini-batch, modelin eğitiminde kullanılır. Her bir deneyim için:

→ **Tahmin Edilen Q-Değerleri:** Sinir ağı, mevcut durum için aksiyonların Q-değerlerini tahmin eder.

→ **Hedef Q-Değerleri:** Bellman denklemi ile güncellenmiş hedef Q-değerleri hesaplanır.

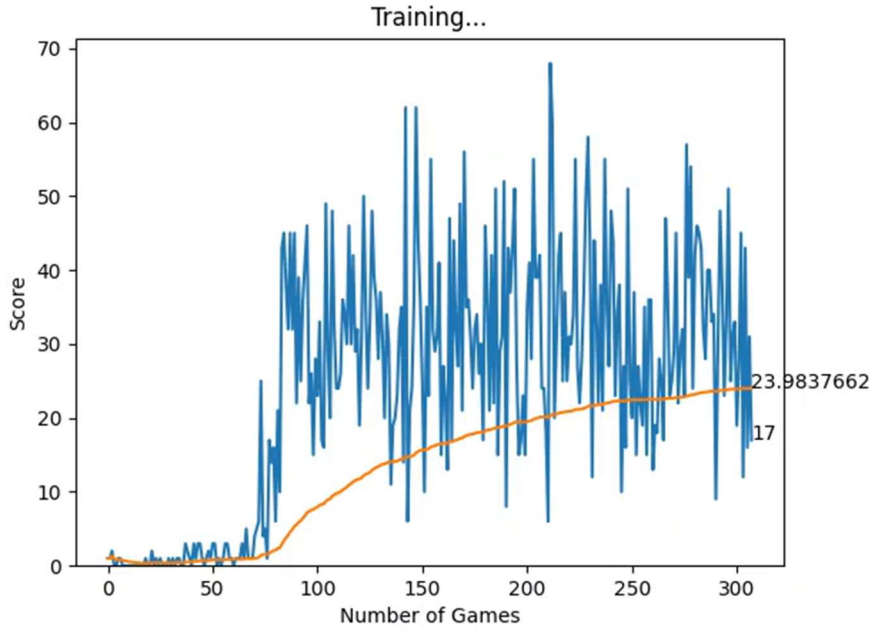
→ **Kayıp Hesaplama:** Tahmin edilen ve hedef Q-değerleri arasındaki fark, kayıp fonksiyonu (örneğin, MSE) ile hesaplanır.

→ **Ağırlık Güncellemesi:** Optimizasyon algoritması (örneğin, Adam) ile sinir ağı güncellenir.

aşamaları uygulanmaktadır.

3. SONUÇ

Bu çalışmada, Deep Q-Network (DQN) algoritması kullanılarak Snake oyununda bir yapay zeka ajanı eğitilmiştir. Ajan, epsilon-greedy politikası ve deneyim tekrar mekanizması ile durum-aksiyon çiftleri arasında ilişkiler öğrenmiş ve bu sayede oyun boyunca performansını artırmıştır. RL DQN tabanlı ajanın eğitim süresince oyunlarda elde ettiği skor değerleri verilmektedir.



Şekil 2. RL DQN Agent Snake Game Score Grafiği

Başlangıçta, ajanın düşük performans sergilediği gözlemlenmiştir. Bu durum, ajanın çevreyi keşfetmeye çalıştığı keşif aşamasını temsil etmektedir. Eğitim ilerledikçe, ajan çevreyi anlamaya başlamış, daha iyi aksiyonlar seçmiş ve bu sayede puanlarını artırmıştır. Ortalama skor grafiğinde sürekli artış eğilimi, ajanın öğrenme kapasitesini ve modelin etkinliğini doğrulamaktadır. Özellikle 150. oyundan sonra, ajanın performansında gözle görülür bir iyileşme olduğu ve öğrenme sürecinin başarılı bir şekilde gerçekleştiği tespit edilmiştir.

Bununla birlikte, grafik üzerinde gözlemlenen dalgalanmalar, epsilon-greedy politikasının etkisiyle ajan davranışının zaman zaman keşif odaklı olduğunu göstermektedir. Bu, ajanın genel başarısına zarar vermeden yeni durumları öğrenmesini sağlamıştır. Dalgalanmalar, RL'nin doğasında bulunan keşif ve sömürü arasındaki dengeyi yansıtmaktadır.

Sonuç olarak, bu proje kapsamında geliştirilen DQN tabanlı ajan, Snake oyununda çevreyi öğrenme ve başarılı bir şekilde kontrol etme yeteneğini kazanmıştır. Eğitim süreci boyunca gözlemlenen performans artışı, yöntemin etkinliğini doğrulamış ve projenin amacına ulaştığını göstermiştir. Bu çalışmanın gelecekte daha karmaşık oyun ortamlarına ve farklı RL yöntemlerine uygulanabilirliği araştırılabilir. Örneğin, farklı ödül mekanizmalarının denenmesi, epsilon azaltma stratejilerinin optimize edilmesi veya daha derin sinir ağı mimarilerinin kullanılması gibi iyileştirmeler bu yöntemin performansını daha da artırabilir.

4. KAYNAK KODLAR

Github Linki: https://github.com/Rexoes/DeepLearning_RL_DQN_Snake

Ekler: RL DQN Snake Game video.

game.py

```
import pygame
import random
from enum import Enum
from collections import namedtuple
import numpy as np

pygame.init()
font = pygame.font.Font('arial.ttf', 25)

class Direction(Enum):
    RIGHT = 1
    LEFT = 2
    UP = 3
    DOWN = 4

Point = namedtuple('Point', 'x, y')

# rgb colors
WHITE = (255, 255, 255)
RED = (200, 0, 0)
BLUE1 = (0, 0, 255)
BLUE2 = (0, 100, 255)
BLACK = (0, 0, 0)

BLOCK_SIZE = 20
SPEED = 40

class SnakeGameAI:

    def __init__(self, w=640, h=480):
        self.w = w
        self.h = h
        # init display
        self.display = pygame.display.set_mode((self.w, self.h))
        pygame.display.set_caption('DQN Snake Game')
        self.clock = pygame.time.Clock()
        self.reset()

    def reset(self):
        # init game state
        self.direction = Direction.RIGHT

        self.head = Point(self.w / 2, self.h / 2)
        self.snake = [self.head,
                      Point(self.head.x - BLOCK_SIZE, self.head.y),
                      Point(self.head.x - (2 * BLOCK_SIZE), self.head.y)]

        self.score = 0
        self.food = None
        self._place_food()
```

```
        self.frame_iteration = 0

    def _place_food(self):
        x = random.randint(0, (self.w - BLOCK_SIZE) // BLOCK_SIZE) * BLOCK_SIZE
        y = random.randint(0, (self.h - BLOCK_SIZE) // BLOCK_SIZE) * BLOCK_SIZE
        self.food = Point(x, y)
        if self.food in self.snake:
            self._place_food()

    def play_step(self, action):
        self.frame_iteration += 1
        # 1. collect user input
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()

        # 2. move
        self._move(action)  # update the head
        self.snake.insert(0, self.head)

        # 3. check if game over
        reward = 0
        game_over = False
        if self.is_collision() or self.frame_iteration > 100 * len(self.snake):
            game_over = True
            reward = -10
            return reward, game_over, self.score

        # 4. place new food or just move
        if self.head == self.food:
            self.score += 1
            reward = 10
            self._place_food()
        else:
            self.snake.pop()

        # 5. update ui and clock
        self._update_ui()
        self.clock.tick(SPEED)
        # 6. return game over and score
        return reward, game_over, self.score

    def is_collision(self, pt=None):
        if pt is None:
            pt = self.head
        # hits boundary
        if pt.x > self.w - BLOCK_SIZE or pt.x < 0 or pt.y > self.h - BLOCK_SIZE or pt.y < 0:
            return True
        # hits itself
        if pt in self.snake[1:]:
            return True

        return False

    def _update_ui(self):
        self.display.fill(BLACK)
```

```
        for pt in self.snake:
            pygame.draw.rect(self.display,   BLUE1,   pygame.Rect(pt.x,   pt.y,   BLOCK_SIZE,
BLOCK_SIZE))
            pygame.draw.rect(self.display, BLUE2, pygame.Rect(pt.x + 4, pt.y + 4, 12, 12))

        pygame.draw.rect(self.display, RED, pygame.Rect(self.food.x, self.food.y, BLOCK_SIZE,
BLOCK_SIZE))

        text = font.render("Score: " + str(self.score), True, WHITE)
        self.display.blit(text, [0, 0])
        pygame.display.flip()

    def _move(self, action):
        # [straight, right, left]

        clock_wise = [Direction.RIGHT, Direction.DOWN, Direction.LEFT, Direction.UP]
        idx = clock_wise.index(self.direction)

        if np.array_equal(action, [1, 0, 0]):
            new_dir = clock_wise[idx]  # no change
        elif np.array_equal(action, [0, 1, 0]):
            next_idx = (idx + 1) % 4
            new_dir = clock_wise[next_idx]  # right turn r -> d -> l -> u
        else:  # [0, 0, 1]
            next_idx = (idx - 1) % 4
            new_dir = clock_wise[next_idx]  # left turn r -> u -> l -> d

        self.direction = new_dir

        x = self.head.x
        y = self.head.y
        if self.direction == Direction.RIGHT:
            x += BLOCK_SIZE
        elif self.direction == Direction.LEFT:
            x -= BLOCK_SIZE
        elif self.direction == Direction.DOWN:
            y += BLOCK_SIZE
        elif self.direction == Direction.UP:
            y -= BLOCK_SIZE

        self.head = Point(x, y)
```

---

agent.py

```
import torch
import random
import numpy as np
from collections import deque
from game import SnakeGameAI, Direction, Point
from model import Linear_QNet, QTrainer
from helper import plot

MAX_MEMORY = 100_000
BATCH_SIZE = 1000
LR = 0.001
```

```
class Agent:

    def __init__(self):
        self.n_games = 0
        self.epsilon = 0  # randomness
        self.gamma = 0.9  # discount rate
        self.memory = deque(maxlen=MAX_MEMORY)  # popleft()
        self.model = Linear_QNet(11, 256, 3)
        self.trainer = QTrainer(self.model, lr=LR, gamma=self.gamma)

    def get_state(self, game):
        head = game.snake[0]
        point_l = Point(head.x - 20, head.y)
        point_r = Point(head.x + 20, head.y)
        point_u = Point(head.x, head.y - 20)
        point_d = Point(head.x, head.y + 20)

        dir_l = game.direction == Direction.LEFT
        dir_r = game.direction == Direction.RIGHT
        dir_u = game.direction == Direction.UP
        dir_d = game.direction == Direction.DOWN

        state = [
            # Danger straight
            (dir_r and game.is_collision(point_r)) or
            (dir_l and game.is_collision(point_l)) or
            (dir_u and game.is_collision(point_u)) or
            (dir_d and game.is_collision(point_d)),

            # Danger right
            (dir_u and game.is_collision(point_r)) or
            (dir_d and game.is_collision(point_l)) or
            (dir_l and game.is_collision(point_u)) or
            (dir_r and game.is_collision(point_d)),

            # Danger left
            (dir_d and game.is_collision(point_r)) or
            (dir_u and game.is_collision(point_l)) or
            (dir_r and game.is_collision(point_u)) or
            (dir_l and game.is_collision(point_d)),

            # Move direction
            dir_l,
            dir_r,
            dir_u,
            dir_d,

            # Food location
            game.food.x < game.head.x,  # food left
            game.food.x > game.head.x,  # food right
            game.food.y < game.head.y,  # food up
            game.food.y > game.head.y  # food down
        ]
        return np.array(state, dtype=int)

    def remember(self, state, action, reward, next_state, done):
```

```
        self.memory.append((state, action, reward, next_state, done))  # popleft if MAX_MEMORY
is reached

    def train_long_memory(self):
        if len(self.memory) > BATCH_SIZE:
            mini_sample = random.sample(self.memory, BATCH_SIZE)  # list of tuples
        else:
            mini_sample = self.memory

        states, actions, rewards, next_states, dones = zip(*mini_sample)
        self.trainer.train_step(states, actions, rewards, next_states, dones)
        # for state, action, reward, nexrt_state, done in mini_sample:
        #    self.trainer.train_step(state, action, reward, next_state, done)

    def train_short_memory(self, state, action, reward, next_state, done):
        self.trainer.train_step(state, action, reward, next_state, done)

    def get_action(self, state):
        # random moves: tradeoff exploration / exploitation
        self.epsilon = 80 - self.n_games
        # self.epsilon = 50 - self.n_games
        final_move = [0, 0, 0]
        if random.randint(0, 200) < self.epsilon:
            move = random.randint(0, 2)
            final_move[move] = 1
        else:
            state0 = torch.tensor(state, dtype=torch.float)
            prediction = self.model(state0)
            move = torch.argmax(prediction).item()
            final_move[move] = 1

        return final_move

def train():
    plot_scores = []
    plot_mean_scores = []
    total_score = 0
    record = 0
    agent = Agent()
    game = SnakeGameAI()
    while True:
        # get old state
        state_old = agent.get_state(game)

        # get move
        final_move = agent.get_action(state_old)

        # perform move and get new state
        reward, done, score = game.play_step(final_move)
        state_new = agent.get_state(game)

        # train short memory
        agent.train_short_memory(state_old, final_move, reward, state_new, done)

        # remember
        agent.remember(state_old, final_move, reward, state_new, done)
```

```
        if done:
            # train long memory, plot result
            game.reset()
            agent.n_games += 1
            agent.train_long_memory()

            if score > record:
                record = score
                agent.model.save()

            print('Game', agent.n_games, 'Score', score, 'Record:', record)

            plot_scores.append(score)
            total_score += score
            mean_score = total_score / agent.n_games
            plot_mean_scores.append(mean_score)
            plot(plot_scores, plot_mean_scores)


if __name__ == '__main__':
    train()
```

---

model.py

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import os

class Linear_QNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = self.linear2(x)
        return x

    def save(self, file_name='model.pth'):
        model_folder_path = './model'
        if not os.path.exists(model_folder_path):
            os.makedirs(model_folder_path)

        file_name = os.path.join(model_folder_path, file_name)
        torch.save(self.state_dict(), file_name)

class QTrainer:
    def __init__(self, model, lr, gamma):
        self.lr = lr
        self.gamma = gamma
        self.model = model
        self.optimizer = optim.Adam(model.parameters(), lr=self.lr)
        self.criterion = nn.MSELoss()
```

```
    def train_step(self, state, action, reward, next_state, done):
        state = torch.tensor(state, dtype=torch.float)
        next_state = torch.tensor(next_state, dtype=torch.float)
        action = torch.tensor(action, dtype=torch.long)
        reward = torch.tensor(reward, dtype=torch.float)
        # (n, x)

        if len(state.shape) == 1:
            # (1, x)
            state = torch.unsqueeze(state, 0)
            next_state = torch.unsqueeze(next_state, 0)
            action = torch.unsqueeze(action, 0)
            reward = torch.unsqueeze(reward, 0)
            done = (done,)

        # 1: predicted Q values with current state
        pred = self.model(state)

        target = pred.clone()
        for idx in range(len(done)):
            Q_new = reward[idx]
            if not done[idx]:
                Q_new = reward[idx] + self.gamma * torch.max(self.model(next_state[idx]))

            target[idx][torch.argmax(action[idx]).item()] = Q_new

        # 2: Q_new = r + y * max(next_predicted Q value) -> only do this if not done
        # pred.clone()
        # preds[argmax(action)] = Q_new
        self.optimizer.zero_grad()
        loss = self.criterion(target, pred)
        loss.backward()

        self.optimizer.step()
```

---

helper.py

```
import matplotlib.pyplot as plt
from IPython import display

plt.ion()

def plot(scores, mean_scores):
    display.clear_output(wait=True)
    display.display(plt.gcf())
    plt.clf()
    plt.title('Training...')
    plt.xlabel('Number of Games')
    plt.ylabel('Score')
    plt.plot(scores)
    plt.plot(mean_scores)
    plt.ylim(ymin=0)
    plt.text(len(scores)-1, scores[-1], str(scores[-1]))
    plt.text(len(mean_scores)-1, mean_scores[-1], str(mean_scores[-1]))
    plt.show(block=False)
    plt.pause(.1)
```