TU Clausthal

Institut für Wirtschaftswissenschaft
Abteilung für Betriebswirtschaftslehre,
insbesondere Produktion und Logistik

Masterprojektarbeit

Implementierung und Test von Deep-Deterministic-Policy-Gradient-Ansätzen für die Steuerung von Entscheidungen zur Speicherung bzw. zum Verkauf regenerativ erzeugter Elektrizität unter Verwendung der Google TensorFlow- und OpenAI Gym-Framework

Sesso Domtchoueng Eric Dietriche

Matrikelnummer: 504999
16.08.2023

Eingerichet bei:
Prof. Dr. Christoph Schwindt

1 Einleitung

Die dynamische Landschaft der Energiemärkte erfordert innovative Ansätze zur effektiven Steuerung und Vermarktung von Elektrizität. Besonders im Kontext der zunehmenden Integration erneuerbarer Energiequellen steht ein profitmaximierendes Energieunternehmen vor der komplexen Aufgabe, optimale Entscheidungen zur Stromerzeugung, Speicherung und Vermarktung zu treffen. Diese Arbeit widmet sich der präzisen Modellierung und Lösung dieser herausfordernden Problemstellung durch den Einsatz fortschrittlicher Deep-Deterministic-Policy-Gradient-Ansätze (DDPG) unter Verwendung der leistungsstarken TensorFlow- und OpenAI Gym-Frameworks.

In diesem Szenario betrachten wir ein Unternehmen, das Elektrizität aus erneuerbaren Quellen generiert und über einen Speicher mit Wandlungsverlusten verfügt. Diese Entscheidungsfindung findet in einem zweigeteilten Marktumfeld statt: dem Intraday-Markt, auf dem Vorabverpflichtungen eingegangen werden, und dem Regulierungsmarkt, der bei unvorhergesehenen Produktionsausfällen zur Verfügung steht. Die zentrale Herausforderung besteht darin, wie das Unternehmen seine Ressourcen strategisch einsetzen kann, um die Verpflichtungen auf dem Intraday-Markt zu erfüllen, unerwartete Produktionslücken auszugleichen und gleichzeitig den Gewinn zu maximieren (Gönsch and Hassler, 2016).

Insbesondere konzentrieren wir uns darauf, wie die DDPG-Ansätze in dieser komplexen Entscheidungslandschaft eingesetzt werden können. Wir nutzen die umfassenden Möglichkeiten der TensorFlow- und OpenAI Gym-Frameworks, um diese anspruchsvolle Aufgabe zu bewältigen. Durch die Integration von DDPG in diesen Rahmen ermöglichen wir eine präzise Modellierung und Lösung des Problems der Stromerzeugung, Speicherung und Vermarktung unter Berücksichtigung von Unsicherheiten, Preisschwankungen und Strafzahlungen.

Die vorliegende Arbeit zielt darauf ab, nicht nur eine tiefgreifende Analyse der Problemstellung und der angewandten Ansätze zu liefern, sondern auch wertvolle Erkenntnisse über die Wirksamkeit von DDPG und die Leistungsfähigkeit der TensorFlow- und OpenAI Gym-Frameworks im Kontext komplexer energiewirtschaftlicher Entscheidungsprozesse zu gewinnen.

2 Steuerung von Entscheidungen bezüglich der Speicherung oder des Verkaufs von regenerativ erzeugter Elektrizität

In diesem Abschnitt stellen wir die mathematische Beschreibung der Umgebung vor, in dem Entscheidungen zur Speicherung oder zum Verkauf von erneuerbar erzeugtem Strom aus Windenergieanlagen getroffen werden. Wir beziehen uns hier auf die spezifischen Parameter und Rahmenbedingungen, wie sie in der Publikation von (Gönsch and Hassler, 2016). beschrieben sind, um das problematische System näher zu charakterisieren.Das System wird diskret zu jedem Zeitpunkt t betrachtet:

Stromerzeugung durch Windturbinen

- **Effizienz bei Speicherung und Entnahme**: Ein wichtiger Aspekt bei der Verwendung von Speichergeräten in einem solchen Szenario sind die Effizienzen beim Speichern und Entnehmen von Energie. Diese Leistungen werden als ρ_R für den Füllungsgrad und ρ_E für den Entfüllungsgrad bezeichnet.Der Speicher- und Entnahmeeffizienz kommt bei der Nutzung von Stromspeichern eine entscheidende Bedeutung zu. Er beschreibt den Prozentsatz der Energie, die beim Laden des Speichers im Verhältnis zur zugeführten Energie effektiv gespeichert wird. Das bedeutet rechnerisch, dass beim Laden einer bestimmten Energiemenge nur ein Teil dieser Energie tatsächlich im Speicher gespeichert wird, während der Rest durch Verluste in Form von Wärme oder anderen Faktoren verloren geht. Ähnliches gilt für die Entladewirkungsgrade, bei denen ebenfalls ein Teil der gespeicherten Energie verloren geht, wenn sie aus dem Speicher entnommen wird. Diese Effizienzwerte sind von entscheidender Bedeutung, da sie den tatsächlichen Energieverlust beim Laden und Entladen quantifizieren und somit den gesamten Energiefluss des Systems beeinflussen.

- **Windgeschwindigkeitsabhängige Parameter**:
  - s_{ci}: Die Einschaltgeschwindigkeit ist die Mindestwindgeschwindigkeit,

bei der die Windturbine mit der Energieerzeugung beginnt. Unterhalb dieser Geschwindigkeit ist die Stromerzeugung nicht ausreichend, um wirtschaftlich rentabel zu sein.

  - s_{co}: Die Abschaltgeschwindigkeit ist die maximale Windgeschwindigkeit, bei der die Anlage aus Sicherheitsgründen abgeschaltet wird. Hohe Windgeschwindigkeiten können zu Belastungen führen, die die Sicherheit der Anlage gefährden.
  - s_r: Die Nenngeschwindigkeit ist die Geschwindigkeit, bei der die Turbine ihre Nennleistung erreicht. Diese Drehzahl gibt den optimalen Betriebspunkt der Turbine an.
  - r : Die Nennleistung der Windkraftanlage gibt an, wie viel Energie sie bei Nennwindgeschwindigkeit erzeugt. Dieser Wert ist ein wesentliches Maß für die Leistungsfähigkeit der Turbine.

- **Länge des Zeitintervalls** Δ_t: Δ_t beeinflusst, wie präzise oder grob die Energieproduktion abgetastet und berechnet wird. Dies wirkt sich auf die Analysegenauigkeit und die Fähigkeit aus, kurzfristige Schwankungen oder langfristige Trends in der Energieerzeugung zu erfassen.Ein kleines δ_t würde bedeuten, dass eine kleine Energiemenge in kürzeren Zeitintervallen berechnet wird. Andererseits würde ein größeres δ_t dazu führen, dass eine große Energiemenge für das gesamte Zeitintervall berechnet wird.

- **Windgeschwindigkeitswerte** : Die im Rahmen dieser Arbeit untersuchten Windgeschwindigkeitswerte WS_t sind über diskrete Zeitpunkte t verteilt und folgen einer statistischen Verteilung, die unter dem Namen Weibull-Verteilung bekannt ist. Diese Verteilung ist durch zwei Parameter gekennzeichnet: den skalaren Parameter λ und den Formparameter k. λ kann als die durchschnittliche Windgeschwindigkeit bezeichnet werden, und k bestimmt, wie steil oder flach die Verteilung um die Nennwindgeschwindigkeit verläuft.

- **Energieerzeugung** Y_t: Die Formel (Gönsch and Hassler, 2016) zur Berechnung der Energieerzeugungswerte Y_t über diskrete Zeitpunkte t einer Windkraftanlage basiert auf verschiedenen Windgeschwindigkeitsbereichen. Der Windgeschwindigkeitswert WS_t wird mit den Schwellenwerten s_{ci} , s_{co} und s_r verglichen, die bestimmte Geschwindigkeitsbereiche darstellen.

  - Wenn der Windgeschwindigkeitswert WS_t kleiner als s_{ci} oder größer als oder gleich s_{co}, wird angenommen, dass die Energieproduktion gleich Null ist, da die Windturbine entweder noch nicht angelaufen ist oder aus Sicherheitsgründen abgeschaltet wurde.

- Wenn der Windgeschwindigkeitswert WS_t im Bereich zwischen s_{ci} und s_r liegt, wird die Energieproduktion nach der Formel (a+b·WS_t^3)·δ_t berechnet. Dabei stehen a und b für Koeffizienten, die durch die Gleichungen a + b·$s_{ci}^3 = 0$ und a +b·s_r^3=r bestimmt werden. Das Ergebnis dieser Berechnung spiegelt die Energiemenge wider, die innerhalb des Zeitintervalls δ_t erzeugt wird.
- Liegt der Windgeschwindigkeitswert WS_t im Bereich zwischen s_r und s_{co}, wird die Energieerzeugung nach der Formel $r \cdot \delta_t$ berechnet. Dies bedeutet, dass die Turbine ihre maximale Leistung bei Windgeschwindigkeiten im Bereich zwischen s_r und s_{co} erreicht und kontinuierlich Energie erzeugt.

Entwicklung der Preise

- **Verkaufpreis** P_t :Der Verkaufspreis für den erzeugten Strom auf dem Intraday-Markt zum Zeitpunkt t wird durch den Ornstein-Uhlenbeck-Prozess modelliert, der auf sinnvolle Weise die Preisschwankungen berücksichtigt. Dieser Prozess kann als mathematische Darstellung eines Preises betrachtet werden, der sich um einen Mittelwert bewegt und stochastischen Schwankungen unterliegt:

$$P_{t+1} = \kappa \cdot \mu + (1 - \kappa) \cdot p_t + \epsilon$$

  Dabei beeinflusst κ die Geschwindigkeit, mit der sich der Preis dem Durchschnittspreis μ annähert. ϵ ist ein zufälliger Störfaktor, der die zufälligen Schwankungen des Preises darstellt. Er wird mit einer Normalverteilung erzeugt, deren Mittelwert 0 und die Standardabweichung σ beträgt.

- **Strafpreis** Q_t:Der Strafpreis für den Kauf von Energie auf dem Regelenergiemarkt zum Zeitpunkt t stellt die zusätzlichen Kosten dar, die entstehen, wenn nicht genügend Energie erzeugt oder aus dem Speicher entnommen wird, um die Verpflichtung zum Zeitpunkt $t - \tau$ (τ steht für die maximal zulässige Lieferzeit für eine verkaufte Strommenge) zu erfüllen. Dieser Strafpreis wird als ein Vielfaches m des aktuellen Verkaufspreises P_t zum Zeitpunkt t festgelegt.Mit dieser Maßnahme sollen Anreize geschaffen werden, um sicherzustellen, dass genügend Energie zur Verfügung steht, um die Verpflichtungen fristgerecht zu erfüllen und mögliche Strafkosten zu vermeiden.

Verpflichtung x_t : Die Entscheidung des Produzenten zum Zeitpunkt t über die Verpflichtung. Bei dem Problem wird davon ausgegangen, dass die Werte kontinuierlich sind und zwischen 0 und einem vordefinierten Maximalwert liegen.

Entwicklung des Füllstands des Speichergeräts L_t: Liegt die Verpflichtung $x_{t-\tau}$ unter dem aktuellen Produktionsniveau Y_t, wird der Füllstand unter Berücksichtigung der Effizienz der Beladung ρ_R angepasst. Dabei wird darauf geachtet, dass der Füllstand die maximale Lagerkapazität nicht überschreitet. Andernfalls, wenn die Verpflichtung $x_{t-\tau}$ größer ist, wird der Füllstand unter Berücksichtigung des Wirkungsgrades beim Entladen ρ_E angepasst. Es wird geprüft, ob ein Energieüberschuss vorhanden ist, und der Füllstand wird entsprechend reduziert, um die Verpflichtung zu erfüllen.

Verfahren der Kontrollentscheidung

Zum Zeitpunkt t trifft der Stromerzeuger eine Entscheidung über die Verpflichtung auf der Grundlage des resultierenden Füllstands L_t , der letzten $\tau - 1$ Verpflichtungen und des Marktpreises P_t. Nach dieser Entscheidung wird geprüft, ob die resultierende Energiemenge aus dem Speicher L_t und die erzeugte Strommenge $Y_t$zum Zeitpunkt $t - \tau$ ausreichen, um die Verpflichtung $x_{t-\tau}$ zu erfüllen. Reicht der Füllstand nicht aus, um die Verpflichtung zu erfüllen, wird der Füllstand nach dem zuvor beschriebenen Mechanismus der Füllstandsentwicklung des Speichers angepasst. Andernfalls wird die zusätzlich benötigte Strommenge automatisch auf dem Regelenergiemarkt zum Preis Q_t gekauft.Das grundlegende Problem besteht darin, innerhalb eines bestimmten Zeithorizonts eine Reihe aufeinanderfolgender Verpflichtungsentscheidungen zu treffen, um den Gesamtprofit des Produzenten auf dem Intraday-Markt zu maximieren.

Markov-Entscheidungsprozess (MDP) für unsere Problemstellung

In Bezug auf unser Fallproblem kann der Zustand s als eine Interaktion von $\tau+1$ Variablen dargestellt werden. Das Zustandstupel s $= (L_t, x_{t-\tau+1}, x_{t-\tau+2}, x_{t-\tau+3}, ..., x_{t-1}, P_t)$ enthält Informationen über den aktuellen Füllstand des Speichers L_t, die vergangenen Verpflichtungen von $x_{t-tau+1}$ zu x_{t-1} und den aktuellen Marktpreis P_t.

Die Aktion a_t repräsentiert die getroffene Verpflichtungsentscheidung x_t zum Zeitpunkt t

Die Übergangsfunktion T beschreibt den Übergang vom aktuellen Zustand S_t zum nächsten Zustand S_{t+1} auf der Grundlage der aktuellen Aktion a_t und der zukünftigen exogenen Einflussgrößen P_{t+1}, Y_{t+1} und Q_{t+1}. Mathematisch gesehen ergibt sich der nächste Zustand S_{t+1} aus dem aktuellen Zustand St und den zu diesem Zeitpunkt getroffenen Maßnahmen sowie aus den zukünftigen Einflussgrößen P_{t+1}, Y_{t+1} und Q_{t+1}:

$$S_{t+1} = T(S_t, a_t, Y_{t+1}, P_{t+1}, Q_{t+1}) = (L_{t+1}, x_{t-+2}, ..., x_t, P_{t+1})$$

Dabei wird die Aktion a_t auf den aktuellen Zustand St angewendet, und die zukünftigen exogenen Variablen P_{t+1}, Y_{t+1} und Q_{t+1} werden verwendet, um die erwarteten Strafkosten zu ermitteln, die in der Zukunft durch die getroffene Entscheidung entstehen können. Die Übergangsfunktion ermöglicht somit die Berechnung des Zustandsübergangs in Abhängigkeit von der aktuellen Entscheidung, den erwarteten zukünftigen Ereignissen und den Auswirkungen auf die relevanten Zustandsvariablen.

Die Belohnung R_t stellt den wirtschaftlichen Erlös aus der Entscheidung für die Verpflichtung x_t dar. Sie setzt sich aus zwei Hauptkomponenten zusammen: dem Erlös aus dem Verkauf der Verpflichtung und den möglichen Sanktionen. Mathematisch ausgedrückt, ist die Belohnung:

$$R_t = P_t \cdot x_t - E[Q_{t+1}[x_{t-\tau+1} - (\rho_E \cdot L_t + Y_{t+1})]^+ | S_t, x_t]$$

In diesem Zusammenhang ermöglicht die Belohnungsfunktion eine umfassende Bewertung der Entscheidungsqualität, indem sie den wirtschaftlichen Ertrag der getroffenen Verpflichtungsentscheidung x_t quantifiziert. Sie berücksichtigt verschiedene Faktoren, wie z.B. den Erlös, der durch den Verkauf von x_t zu einem bestimmten Marktpreis P_t erzielt wird, die potenziellen Strafen Q_{t+1} für den Kauf einer Einheit aus der Strommenge auf dem Regulierungsmarkt zu einem späteren Zeitpunkt (t+1). Dies geschieht für den Fall, dass der effiziente Energieentnahme $\rho_E \cdot L_t$ aus dem resultierenden Speicherfüllstand L_t zum Zeitpunkt t und die zukünftige Energieerzeugung Y_{t+1} zum Zeitpunkt t+1 für die Lieferung der Verpflichtung $x_{t-\tau+1}$ nicht ausreichen. Die Belohnungsfunktion ermöglicht somit eine umfassende Bewertung, da sie sowohl die kurzfristigen als auch die langfristigen Auswirkungen der Entscheidung berücksichtigt. Sie basiert auf einer Kombination aus aktuellen Werten und erwarteten zukünftigen Entwicklungen, wobei der Diskontierungsfaktor β die Auswirkungen der zeitlichen Distanz moduliert. Auf diese Weise kann der Erzeuger die Verpflichtungen effektiv verwalten, um ein optimales Gleichgewicht zwischen der Erfüllung der Verpflichtungen und der wirtschaftlichen Rentabilität zu erreichen.

3 Deep deterministic Policy Gradient (DDPG)

Der Deep Deterministic Policy Gradient (DDPG) Algorithmus ist eine Alternative für die Lösung von Entscheidungsproblemen in Entscheidungsproblemen in kontinuierlichen Aktionsräumen.(Lillicrap et al., 2015)

Algorithm 1 Deep Deterministic Policy Gradient (DDPG)

1: Initialisiere das Kritiker-Netzwerk $Q(s, a|\theta^Q)$ und das Aktoren-Netzwerk $\mu(s|\theta^\mu)$ mit Gewichten θ^Q und θ^μ.
2: Initialisiere die Ziel-Netzwerke Q' und μ' als Kopien der ursprünglichen Netzwerke mit Gewichten $\theta^{Q'} \leftarrow \theta^Q$ und $\theta^{\mu'} \leftarrow \theta^\mu$.
3: Initialisiere den Replay-Puffer R.
4: **for** Episode = 1 bis M **do**
5: Initialisiere einen zufälligen Prozess N für die Aktionsexploration.
6: Empfange den anfänglichen Beobachtungszustand s_1.
7: **for** Zeitschritt $t = 1$ bis T **do**
8: Wähle die Aktion $a_t = \mu(s_t|\theta^\mu) + N_t$ gemäß der aktuellen Richtlinie und Exploration.
9: Führe die Aktion a_t aus, erhalte die Belohnung r_t und den neuen Zustand s_{t+1}.
10: Speichere den Übergang (s_t, a_t, r_t, s_{t+1}) im Replay-Puffer R.
11: Wähle eine zufällige Minicharge von N Übergängen (s_i, a_i, r_i, s_{i+1}) aus R.
12: Berechne die Ziel-Q-Werte $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$.
13: Aktualisiere das Kritiker-Netzwerk $Q(s_i, a_i|\theta^Q)$ durch Minimierung des Verlusts $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$.
14: Aktualisiere das Aktoren-Netzwerk $\mu(s|\theta^\mu)$ mit der Richtlinie $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$.
15: Aktualisiere die Ziel-Netzwerke: $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$ und $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$.
16: **end for**
17: **end for**

Der DDPG-Algorithmus beginnt mit der Initialisierung von Kritiker- und Akteursnetzen mit Zufallsgewichten (Zeile 1). Darüber hinaus werden die Zielnetze aus Stabilitätsgründen initialisiert, indem Kopien der ursprünglichen Netze erstellt werden (Zeile 2). Es wird ein Wiederholungspuffer angelegt, um Erfahrungen zu speichern (Zeile 3). Der Algorithmus durchläuft eine Reihe von Episoden (Schleife in Zeile 4), in denen der Agent agiert. Die zufällige Erkundung wird eingeführt (Zeile 5) und der Anfangszustand der Episode wird empfangen (Zeile 6). Bei jedem Zeitschritt (Schleife in Zeile 6) wird eine Aktion entsprechend der Aktionspolitik (Zeile 7) ausgewählt und in der Umgebung ausgeführt. Die daraus resultierende Belohnung und der nächste Zustand werden beobachtet (Zeilen 8-9) und im Wiedergabepuffer gespeichert (Zeile 10). Es werden zufällige Ministapel von Übergängen genommen (Zeile 11), um die Q-Zielwerte unter Verwendung der Bellman-Gleichung zu berechnen (Zeile 11). Das Kritikernetz wird aktualisiert, um den Verlust zu minimieren (Zeile 12), und das Akteursnetz wird mithilfe von Policy-Descent-Techniken verbessert (Zeile 13). Die Zielnetzwerke werden aktualisiert, um stabile Zielschätzungen zu erhalten (Zeilen 14-15). Der Prozess wird für alle Episoden wiederholt (Schleife in Zeile 4) und endet nach Durchlaufen aller Episoden (Zeile 17).

4 Implementierung von DDPG-Algorithmen unter Verwerdung von TensorFlow- und OpenAI Gym-Frameworks

In diesem Abschnitt möchte ich die praktische Umsetzung meiner Arbeit vorstellen. Dazu zeige ich Codefragmente in Bildformat, gefolgt von einer kurzen Erläuterung, welche spezifischen Funktionen sie jeweils realisieren. Dies wird einen detaillierten Einblick in den Implementierungsprozess des DDPG-Algorithmus geben, der mit den Frameworks TensorFlow und OpenAI Gym entwickelt wurde.

DDPG-Tensorflow

```
1     class DDPGAgent:
2         def __init__(self, state_shape, action_shape, action_bound):
3             self.state_shape = state_shape
4             self.action_shape = action_shape
5             self.action_bound = action_bound
6             self.actor_lr = 1e-5
7             self.critic_lr = 1e-4
8             self.tau = 1e-4
9             self.gamma = 0.95
10
11            self.actor = self.build_actor_model()
12            self.critic = self.build_critic_model()
13
14            self.target_actor = self.build_actor_model()
15            self.target_critic = self.build_critic_model()
16
17            actor and the critic weights, respectively.
18            self.target_actor.set_weights(self.actor.get_weights())
19            self.target_critic.set_weights(self.critic.get_weights())
20
```

```
21              self.critic_loss = MeanSquaredError()
22              self.actor_optimizer = Adam(learning_rate=self.actor_lr)
23              self.critic_optimizer = Adam(learning_rate=self.critic_lr)
24              self.buffer_size = 1000000
25              self.memory = deque(maxlen=self.buffer_size)
```

Das vorliegende Codefragment repräsentiert den Konstruktor der DDPGAgent-Klasse, welcher beim Initiieren eines DDPG-Agents aufgerufen wird. Innerhalb dieses Konstruktors werden grundlegende Einstellungen vorgenommen, die den Agenten in seiner Lernumgebung definieren. Dabei werden sogenannte Hyperparameter, die wie Steuerelemente für den Lernprozess sind, festgelegt. Diese umfassen Dinge wie die Form der Zustände (Informationen über die Umgebung) und Aktionen (Entscheidungen des Agenten). Darüber hinaus werden Netzwerkstrukturen erstellt, die das Verhalten des Agenten steuern – der sogenannte Äktor"gibt Handlungsanweisungen, während der "Kritiker"diese Aktionen bewertet. Um den Lernprozess effizienter zu gestalten, werden auch spezielle SZiel-Netzwerkeërzeugt, die Kopien der ursprünglichen Netzwerke sind. Um die Agenten-Entscheidungen zu optimieren, wird eine Verlustfunktion für den Kritiker definiert und Optimierer mit passenden Lernraten festgelegt. Zusätzlich wird ein Wiedergabepuffer erstellt, der dazu dient, Erfahrungen aus vorherigen Handlungen zu speichern und später für das Training zu verwenden. Zusammengefasst etabliert dieser Konstruktor die grundlegenden Parameter und Strukturen, die für den DDPG-Algorithmus und die Umsetzung des Agenten unerlässlich sind.

```
1       def build_actor_model(self):
2           inputs = Input(shape=self.state_shape)
3           x = Dense(128, activation='relu')(inputs)
4           x = Dense(64, activation='relu')(x)
5           x = Dense(32, activation='relu')(x)
6           x = Dense(16, activation='relu')(x)
7           x = Dense(32, activation='relu')(x)
8           x = Dense(64, activation='relu')(x)
9           x = Dense(128, activation='relu')(x)
10          output = Dense(self.action_shape[0], activation='sigmoid')(x)
11          output = output*self.action_bound
12          model = Model(inputs, output)
13          return model
14
15      def build_critic_model(self):
16          state_input = Input(shape=self.state_shape)
17          action_input = Input(shape=self.action_shape)
18          x = Concatenate()([state_input, action_input])
```

```
19          x = Dense(128, activation='relu')(x)
20          x = Dense(64, activation='relu')(x)
21          x = Dense(32, activation='relu')(x)
22          x = Dense(16, activation='relu')(x)
23          x = Dense(32, activation='relu')(x)
24          x = Dense(64, activation='relu')(x)
25          x = Dense(128, activation='relu')(x)
26          output = Dense(1)(x)
27          model = Model([state_input, action_input], output)
28          return model
```

Die Funktion `build_actor_model` erstellt das Akteursmodell in einem akteurskritischen neuronalen Netz. Die Eingabeschicht besteht aus `self.state_shape` Neuronen, die den Zustand des Akteurs darstellen. Darauf folgen mehrere Schichten, die nach und nach die Merkmale der Eingabe verarbeiten. Die erste Schicht hat 128 Neuronen, um komplexe Merkmale zu erfassen. Die anderen Schichten sind symmetrisch aufgebaut, d. h. die Anzahl der Neuronen wird abwechselnd reduziert und dann erweitert. Diese Anordnung ermöglicht die schrittweise Extraktion abstrakter Merkmale aus den Daten. Während die Schichten mit weniger Neuronen dazu beitragen, relevante Informationen zu konzentrieren und das Rauschen zu minimieren, ermöglichen nachfolgende Schichten mit einer höheren Neuronenzahl eine progressivere Integration dieser Merkmale für die spätere Generierung und Bewertung von Aktionen.Aber Die Ausgabeschicht besteht genau aus so vielen Neuronen, wie es mögliche Aktionen gibt (`self.action_shape[0]`). Mit Hilfe der Sigmoid-Aktivierungsfunktion werden Werte zwischen 0 und 1 erzeugt. Diese Werte werden dann mit `self.action_bound` multipliziert, um sie in den Aktionsraum der Umgebung zu skalieren.
Die Funktion `build_critic_model` erstellt das kritische Modell, das im neuronalen Netz Actor Critic verwendet wird, und akzeptiert Zustand und Aktion als Eingaben. Diese werden in der Concatenate-Schicht zusammengeführt, um eine umfassende Eingabe für die Verarbeitung zu erstellen. Die Architektur des neuronalen Netzes ähnelt der des Actor-Modells mit schrittweiser Merkmalsextraktion. Die Ausgabeschicht besteht aus einem einzigen Neuron, das dem Dimensionswert des Aktionsraums (`self.action_shape[0]`) entspricht.

```
1 def update_target(self, target_weights, weights):
2     for (target_weight, weight) in zip(target_weights, weights):
3         target_weight.assign(weight*self.tau + target_weight*(1 - self.tau))
```

Die Funktion `update_target` implementiert die Strategie der Soft Update zur schrittweisen Aktualisierung der Gewichte eines neuronalen Zielnetzes. In dieser Funktion werden die Gewichte jeder Schicht im Zielnetz, dargestellt als `target_weights`,

durch eine Mischung aus den Gewichten der entsprechenden Schicht im Quellnetz (weights) und den vorhandenen Zielgewichten verändert. Die Art der Mischung wird durch den Hyperparameter self.tau gesteuert. Dies ermöglicht eine sanfte und kontinuierliche Anpassung der Zielgewichte an die des Quellnetzes, ohne abrupte Änderungen. Dieser Prozess findet für jede Schicht in den Netzen statt und trägt dazu bei, dass das Zielnetz allmählich an die Daten des Quellnetzes angepasst wird.

```
1         def get_action(self, state):
2             state = np.expand_dims(state, axis=0)
3             action = self.actor.predict(state)[0]
4             return action
```

Die Funktion `get_action` zielt auf die Vorhersage zukünftiger Aktionen unter Verwendung des Actor-Netzwerks ab. Der übergebene Zustand wird zuvor für die Netzwerkvorhersage vorbereitet, indem eine Batch-Dimension mit `np.expand_dims` eingefügt wird, was insbesondere in Deep-Learning-Frameworks wie Keras oder TensorFlow üblich ist. Diese Vorbereitung ermöglicht es, auch wenn nur ein Zustand verarbeitet wird, die Konsistenz der Eingabe zu erhalten. Anschließend wird das so aufbereitete Zustandsarray an das Akteursnetzwerk übergeben, um die Aktion vorherzusagen, die der Agent im aktuellen Zustand ausführen sollte. Die Vorhersage wird als Ausgabe des Netzes empfangen und extrahiert, um die beabsichtigte Aktion darzustellen, die in dem gegebenen Zustand erwartet wird.

```
1     def remember(self, state, action, reward, next_state, done):
2         self.memory.append((state, action, reward, next_state, done))
3
4      def sample_batch(self, batch_size):
5         batch = random.sample(self.memory, batch_size)
6         states, actions, rewards, next_states, dones = zip(*batch)
7         return states, actions, rewards, next_states, dones
```

Die Funktion remember wird verwendet, um Erfahrungen in einem Pufferspeicher zu sichern, der als Wiederholungspuffer dient. Jedes Erlebnis wird als Tupel (state, action, reward, `next_state`, done) gespeichert, wobei state den aktuellen Zustand, action die durchgeführte Aktion, reward die erhaltene Belohnung, `next_state` den nächsten Zustand und done einen Wert angibt, ob der Zustand abgeschlossen ist (d. h. das Ende einer Episode darstellt).

Die Funktion `sample_batch` wird verwendet, um zufällig gespeicherte Erlebnisse aus dem Wiedergabepuffer auszuwählen. Dazu wird eine Zufallsstichprobe von Erlebnissen der Größe `batch_size` gezogen. Die Stichprobe wird in fünf separate Listen aufgeteilt: states, actions, rewards, `next_states` und dones. Diese Listen enthalten jeweils die entsprechenden Elemente der ausgewählten Erlebnisse.

```
def sample_batch(self, batch_size):
    batch = random.sample(self.memory, batch_size)
    states, actions, rewards, next_states, dones = zip(*batch)
    return states, actions, rewards, next_states, dones

def train(self, states, actions, next_states, rewards, dones):
    states = np.array(states)
    actions = np.array(actions)
    next_states = np.array(next_states)
    rewards = np.array(rewards)
    dones = np.array(dones)

    with GradientTape() as tape:
        target_actions = self.target_actor(next_states)
        target_Q_values = self.target_critic([next_states, target_actions])
        target_Q_values = rewards + self.gamma*target_Q_values*(1 - dones)
        Q_values = self.critic([states, actions])
        critic_loss = self.critic_loss(target_Q_values, Q_values)

        critic_grads = tape.gradient(critic_loss,
                                     self.critic.trainable_variables)
        self.critic_optimizer.apply_gradients(zip(critic_grads,
                                     self.critic.trainable_variables))

        with GradientTape() as tape:
            predicted_actions = self.actor(states)
            actor_loss = -reduce_mean(self.critic([states,
                                                   predicted_actions]))
        actor_grads = tape.gradient(actor_loss,
                                    self.actor.trainable_variables)
        self.actor_optimizer.apply_gradients(zip(actor_grads,
                                    self.actor.trainable_variables))

        self.update_target(self.target_actor.variables,
                                                self.actor.variables)
        self.update_target(self.target_critic.variables,
                                                    self.critic.variables)

def train_batch(self, batch_size):
  if len(self.memory) < batch_size:
       return
  states, actions, rewards, next_states, dones =self.sample_batch(batch_size)
```

```
43    self.train(states, actions, next_states, rewards, dones)
```

Die train-Funktion übernimmt den Trainingsprozess der Kritiker- und Akteursnetze beim Reinforcement Learning. Zunächst wandelt sie die übergebenen Erfahrungen in NumPy-Arrays um. Dann erfolgt die Aktualisierung der Kritiker: Sie verwendet GradientTape, um die Verlustgradienten der Kritiker zu erfassen, nutzt das `target_actor_Netzwerk`, um Zielaktionen für die nächsten Zustände zu erhalten, und berechnet auf dieser Grundlage die Q-Zielwerte im `target_critic_Netzwerk`. Diese Q-Werte werden in Abhängigkeit von Belohnungen und Abschlüssen aktualisiert. Die aktuellen Q-Werte werden vom kritischen Netz für die Zustands-Aktionspaare berechnet. Dann wird der Kritikerverlust berechnet und die Gewichte des kritischen Netzes werden mit den Verlustgradienten aktualisiert. Bei der Akteursaktualisierung sagt das Akteursnetz Aktionen für Zustände voraus, und im GradientTape-Kontext werden die Gradienten des negativen Durchschnitts der vom Kritikernetz ausgegebenen Q-Werte aufgezeichnet. Ein Akteursverlust wird erzeugt, um Aktionen zu fördern, die höhere Q-Werte erreichen. Schließlich wird mit der Funktion `update_target` eine sanfte Aktualisierung der Zielnetze durchgeführt. Die Funktion `train_batch` stellt sicher, dass genügend Erfahrungen im Speicher vorhanden sind, wählt zufällige Erfahrungen aus und ruft train auf, um die Netze zu trainieren. Dieser Ablauf ermöglicht das schrittweise Lernen und die Optimierung der Netze für ein optimales Verhalten im Kontext des Reinforcement Learning.

Training eines Tensorflow-DDPGAgent

```
1  env = EnergyMarketEnv()
2
3  state_shape = env.observation_space.shape
4  action_shape = env.action_space.shape
5  action_bound = env.action_space.high[0]
6
7  agent = DDPGAgent(state_shape, action_shape, action_bound)
8
9  episodes = 100
10 max_steps = 20
11 batch_size = 50
12 total_score = 0
13
14 for episode in range(episodes):
15     state = env.reset()
16     score = 0
17
18     for step in range(max_steps):
```

```
        # Get the next action using the updated exploration rate
        action = agent.get_action(state)

        next_state, reward, done, _ = env.step(action)
        agent.remember(state, action, reward, next_state, done)

        score += reward
        state = next_state

        if done:
            break

    if len(agent.memory) > batch_size:
        agent.train_batch(batch_size)

    print(f"Episode {episode + 1}, Reward: {score}")
    total_score += score

env.close()
```

Der vorliegende Code zeigt den Trainingsprozess eines Deep Reinforcement Learning Agenten in unserer Energiewirtschaftsumgebung (EnergyMarketEnv). Zunächst werden die Grundlagen gelegt, indem die Umgebung initialisiert wird. Dabei werden nicht nur die Dimensionen des Zustandsraums (`state_shape`) sowie des Aktionsraums (`action_shape`) ermittelt, sondern auch die begrenzenden Werte der Aktionen (`action_bound`) extrahiert. Anschließend wird der Agent, ein DDPGAgent, unter Berücksichtigung dieser Umgebungsmerkmale initialisiert.

Während einer festgelegten Anzahl von Episoden interagiert der Agent mit der Umgebung. Jede Episode beginnt mit einem Neustart des Zustands, gefolgt von der schrittweisen Auswahl von Aktionen, dem Speichern von Erfahrungen und der Akkumulation der erhaltenen Belohnungen (reward). Sobald genügend Erfahrungen angesammelt sind, erfolgt das Training des Agenten durch die `train_batch`-Funktion, wobei Stapel von Erfahrungen mit einer Größe von `batch_size` genutzt werden.

Das Hauptziel dieses Codes ist es, dem Agenten zu ermöglichen, auf Grundlage von Zustandsinformationen zu interagieren, aus Erfahrungen zu lernen und seine Aktionen schrittweise zu optimieren. Die Gesamtpunktzahl (`total_score`) dient als Maß für die Leistung des Agenten über alle Episoden hinweg. Nach Abschluss aller Episoden wird die Umgebung ordnungsgemäß geschlossen. Insgesamt visualisiert dieser Code den iterativen Prozess, bei dem der Agent kontinuierlich sein Verhalten verbessert, um in der Energiewirtschaftsumgebung fundierte Entscheidungen zu treffen und die Gesamtleistung zu maximieren

Training eines Gmy-DDPGAgents

```
env = EnergyMarketEnv()
observation_shape = env.observation_space.shape
nb_actions = env.action_space.shape[0]

# The noise objects for DDPG
n_actions = env.action_space.shape[-1]
action_noise = OrnsteinUhlenbeckActionNoise
                        (mean=np.zeros(env.action_space.shape[0]),
                         sigma=0.1 * np.ones(env.action_space.shape[0]))

model = DDPG("MlpPolicy", env,
             learning_rate=0.001, action_noise=action_noise, verbose=1)
model.learn(total_timesteps=1000, log_interval=10)

vec_env = model.get_env()
episodes = 100
total_score = 0
for episode in range(1, episodes + 1):
    obs = vec_env.reset()
    score = 0
    done = False
    action, _ = model.predict(obs)
    while not done:
        action, _ = model.predict(obs)
        obs, reward, done, _ = vec_env.step(action)
        score += reward[0]
    print(f"Episode {episode + 1}, Reward: {score}")
    total_score += score
print(f"Mean reward: {total_score/episodes}")
```

Der vorliegende Code verwendet die Stable Baselines 3 Bibliothek, um den Deep Deterministic Policy Gradient (DDPG) Algorithmus für die Entscheidungsfindung in unser energiewirtschaftlichen Umgebung einzusetzen. Im Vergleich zur vorherigen Darstellung eines Tensorflow DDPG-Fragments, bei der der Algorithmus manuell implementiert werden musste, bietet die Stable Baselines 3-Bibliothek bereits eine effiziente und vordefinierte Version des DDPG-Algorithmus. Der Code beginnt mit der Initialisierung der Umgebung und dem Abrufen relevanter Informationen über den Zustandsraum und die möglichen Aktionen. Um die Exploration des Aktionsraums zu unterstützen, wird ein Rauschobjekt erzeugt, hier repräsentiert durch OrnsteinUhlenbeckActionNoise. Es ermöglicht eine kontrollierte Variation der Aktionen während des Trainings. Das DDPG-Modell wird erstellt und mit den

notwendigen Parametern wie Lernrate und Aktionsrauschen konfiguriert. Durch Aufruf von learn wird das Modell für eine vorgegebene Anzahl von Zeitschritten trainiert. Nach dem Training wird das Modell in eine Vektorumgebung integriert. Dort werden Episoden ausgeführt. In jeder Episode interagiert der Agent mit der Umgebung, trifft Entscheidungen auf der Grundlage des gelernten Modells und sammelt Belohnungen. Diese Belohnungen werden zur Bestimmung der Gesamtpunktzahl der Episode aufsummiert. Die Leistung des Agenten wird durch die Ausgabe der Episodennummer, der erreichten Punktzahl und der kumulierten Gesamtpunktzahl bewertet.

Die Umgebung für unsere Energiewirtschaft wurde mit den in der Tabelle Parameterwerte initialisiert (Gönsch and Hassler, 2016).

Tabelle 4.1: Verwendete Daten für die Evaluierung der Modells

Parameter	Werte
s_{ci}	3 m/s
s_{co}	25 m/s
s_r	12 m/s
r	20 MW
L_{max}	2.5 MWh
ρ_R	$\sqrt{0.9}$
ρ_E	$\sqrt{0.9}$
T	20
Δ_t	0.25h
β	1
m	2
x_{max}	6.25 MWh
Weibull-Verteillung	
λ	1/0.127
k	1.430
Preis	
κ	1.035
μ	40.712

Der Parameter L_{max} steht für die maximal verfügbare Speicherkapazität unseres Speichers, T für die gesamte Simulationsdauer und x_max für die maximale Strommenge, die der Agent zu jedem Zeitpunkt der Simulation verkaufen kann.

5 Fazit

Diese Projektarbeit baut auf der Forschungsarbeit von den Herren Gönsch und Hassler ((Gönsch and Hassler, 2016)) auf, der ein hybrides approximatives Politikiterationsverfahren (HAPI) zur Lösung des Entscheidungsproblems im Kontext der Speicherung und des Verkaufs von regenerativ erzeugtem Strom entwickelt hat. Als Student habe ich mich intensiv mit dieser Thematik beschäftigt und mich auf einen alternativen Ansatz, den Deep Deterministic Policy Gradient (DDPG), konzentriert.Die Umsetzung des DDPG-Ansatzes erfolgte mit Hilfe der Frameworks TensorFlow und Gym, um eine effiziente Implementierung zu gewährleisten. Ein Hauptziel dieser Arbeit war es zu untersuchen, wie Deep Reinforcement Learning (DRL) Verfahren die Entscheidungsfindung in diesem spezifischen Kontext optimieren können. Die Leistungsfähigkeit beider Ansätze - des etablierten HAPI-Verfahrens von Gönsch und des von mir gewählten DDPG-Ansatzes - wurde anhand von Januar-Datensätzen, wie sie in der Arbeit von Gönsch ausführlich beschrieben sind, evaluiert.Die Ergebnisse zeigen, dass der DDPG-Ansatz vielversprechend ist, auch wenn er im Vergleich zum HAPI-Ansatz etwas niedrigere Profitraten aufweist. Diese Arbeit gibt einen Einblick in die Vielfalt der Entscheidungsansätze in diesem speziellen Anwendungsgebiet und bietet eine Grundlage für zukünftige Forschung und Innovation.

Literaturverzeichnis

Jochen Gönsch and Michael Hassler. Sell or store? an adp approach to marketing renewable energy. *OR spectrum*, 38(3):633–660, 2016.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.