МОНГОЛ УЛСЫН ИХ СУРГУУЛЬ
ХЭРЭГЛЭЭНИЙ ШИНЖЛЭХ УХААН, ИНЖЕНЕРЧЛЭЛИЙН СУРГУУЛЬ
МЭДЭЭЛЭЛ, КОМПЬЮТЕРИЙН УХААНЫ ТЭНХИМ

Батбаярын Бямбаням

DDPG бататгасан сургалтын үйлдлийн шуугианыг турших, харьцуулах

(Effect of action space noise for DDPG RL algorithm)

Мэдээллийн технологи (D061304)
Бакалаврын судалгааны ажил

Улаанбаатар

2021 оны 02 сар

МОНГОЛ УЛСЫН ИХ СУРГУУЛЬ
ХЭРЭГЛЭЭНИЙ ШИНЖЛЭХ УХААН, ИНЖЕНЕРЧЛЭЛИЙН СУРГУУЛЬ
МЭДЭЭЛЭЛ, КОМПЬЮТЕРИЙН УХААНЫ ТЭНХИМ

DDPG бататгасан сургалтын үйлдлийн шуугианыг турших, харьцуулах

(Effect of action space noise for DDPG RL algorithm)

Мэдээллийн технологи (D061304)
Бакалаврын судалгааны ажил

Удирдагч: _____ Г.Гантулга

Хамтран удирдагч: _____

Гүйцэтгэсэн: _____ Б.Бямбаням (17B1NUM1479)

Улаанбаатар

2021 оны 02 сар

Зохиогчийн баталгаа

Миний бие Батбаярын Бямбаням ”DDPG бататгасан сургалтын үйлдлийн шуугианыг турших, харьцуулах ” сэдэвтэй судалгааны ажлыг гүйцэтгэсэн болохыг зарлаж дараах зүйлсийг баталж байна:

- Ажил нь бүхэлдээ эсвэл ихэнхдээ Монгол Улсын Их Сургуулийн зэрэг горилохоор дэвшүүлсэн болно.
- Энэ ажлын аль нэг хэсгийг эсвэл бүхлээр нь ямар нэг их, дээд сургуулийн зэрэг горилохоор оруулж байгаагүй.
- Бусдын хийсэн ажлаас хуулбарлаагүй, ашигласан бол ишлэл, зүүлт хийсэн.
- Ажлыг би өөрөө (хамтарч) хийсэн ба миний хийсэн ажил, үзүүлсэн дэмжлэгийг дипломын ажилд тодорхой тусгасан.
- Ажилд тусалсан бүх эх сурвалжид талархаж байна.

Гарын үсэг: _____

Огноо: _____

ГАРЧИГ

ЗУРГИЙН ЖАГСААЛТ

ХҮСНЭГТИЙН ЖАГСААЛТ

Кодын жагсаалт

УДИРТГАЛ

Сүүлийн жилүүдэд бататгасан сургалтын (RL) алгоритмуудыг компьютерын шинжлэх ухааны судлаачид ихээр сонирхон судалж байна. Ялангуяа Model-free RL алгоритмуудыг олон янзын төвөгтэй асуудлуудыг маш өндөр үр дүнтэйгээр шийдвэрлэхэд ихээр ашиглаж байгаа бөгөөд гүн неорал сүлжээг (DNN) нэмж ашигласнаар үр дүнг илүү сайжруулж байна.

RL алгоритмуудыг судлах, үнэлэх зориулалттай олон төрлийн сангууд бий болсоор байна. Эдгээрийн нэг нь OpenAI бөгөөд энэ нь нээлттэй эх сурвалж бүхий сан юм. Үүнд Gym буюу бататгасан сургалтын алгоритмуудыг хөгжүүлэх, харьцуулахад зориулсан хэрэгсэл, Baseline сан буюу сүүлийн үеийн бататгасан сургалтын алгоритмуудыг Tensorflow ашиглан гүйцэтгэсэн хэрэгжүүлэлт бүхий сан зэрэг багтана. Deep Deterministic Policy Gradient (DDPG) бол Baseline санд хэрэгжүүлэгдсэн алгоритмуудын нэг бөгөөд олон янзын нарийн төвөгтэй даалгаврууд үр дүнтэй шийдвэрлэж чадах алгоритм юм.

Бататгасан сургалтад тасралттай үйлдлийн хувьд суралцах үйл явц нь санамсаргүй үйлдлийг сонгох замаар явагддаг. Харин үргэлжилсэн үйлдлийн хувьд суралцах үйл явц нь үйлдэлд шуугианыг нэмэх замаар явагддаг. DDPG бол тасралтгүй, үргэлжилсэн үйлдлүүдийг сурахад зориулагдсан алгоритм тул шуугиан ашиглагдана. Энэ судалгааны ажлаар 3 төрлийн шуугианыг харьцуулах, турших бөгөөд ямар үр нөлөөтэй, аль шуугиан нь илүү болохыг тодорхойлно.

0.1 Зорилго

DDPG бататгасан сургалтын үйлдлийн шуугианыг туршин, харьцуулж энэ шуугиан моделийг сургах үйл явцад хэрхэн нөлөөлж буйг ажиглан дүгнэлт гаргах

0.2 Асуудал

PyTorch санг ашиглан хэрэгжүүлсэн DDPG алгоритмын үр дүн ямар төрлийн шуугиан ашиглахаас хамааран хэрхэн өөрчлөгдөж байна вэ? Аль шуугиан нь илүү үр дүнтэй байна вэ?

0.3 Хамрах хүрээ

Орчны шуугиануудыг хооронд нь харьцуулж, үнэлэхийн тулд ижил орчинд DDPG алгоритмыг ашиглан хэд хэдэн удаа туршин үзнэ. Энэхүү судалгааны ажлын бүх туршилтыг BipedalWalker-V3 симуляцийн орчинд хийнэ. Мөн бүх шуугианы хувьд mu=0, sigma=0.2 байна. Mu нь дундаж утга, sigma нь савлах утга. [-0.2,0.2] гэсэн хязгаарын дотор шуугианы утгыг авна гэсэн үг.

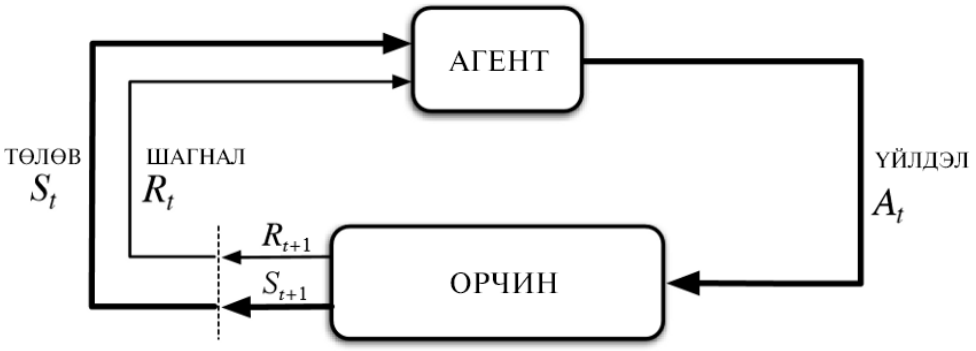
1. СУДАЛГАА

Энэ хэсэгт RL-ийн цаад санаануудыг болон Deep Deterministic Policy Gradient (DDPG) алгоритмыг ойлгоход шаардлагатай зарим онолыг тайлбарлалаа. Мөн ашигласан технологийн тайлбарыг орууллаа.

1.1 Бататгасан сургалтын алгоритм

DDPG алгоритмыг тайлбарлахаас өмнө Reinforcement Learning буюу бататгасан сургалтын талаар бага зэрэг тайлбарлая. Цаашдаа RL гэж товчлон бичнэ.

RL нь агент болон орчин гэсэн хоёр хэсгээс тогтдог. Орчин гэдэг нь агент ажиллаж байгаа объектыг, агент гэдэг нь RL алгоритмыг илэрхийлнэ.



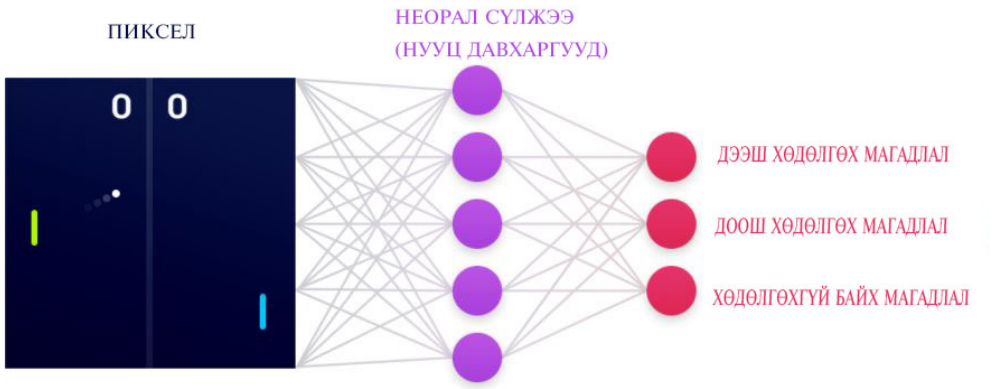
Зураг 1.1: Бататгасан сургалтын бүтэц

Орчин нь агентруу төлөвийг илгээх байдлаар эхэлдэг бөгөөд агент нь мэдлэг дээрээ тулгуурлан тухайн нөхцөл байдалд хариу үйлдэл үзүүлнэ. Үүний дараа орчин агент руу дараагийн төлөв болон reward-ыг илгээнэ. Агент нь орчноос хүлээн авсан reward-аар мэдлэгээ шинэчилнэ. Энэ давталт нь орчноос дуусгах төлөв илгээх хүртэл үргэлжилнэ. Ихэнх RL алгоритмууд дээрх байдлаар ажилладаг.

Агентын төлөв байдлыг дүрслэхийн тулд тухайн нөхцөл байдалд агент хэрхэн ажиллах

ёстойг тооцоолох полиси (policy) функцийг (агентийн тархи гэсэн үг) тодорхойлдог. Практик дээр полиси (policy) нь ихэвчлэн тоглоомын одоогийн төлөвийг оролт болгон авч зөвшөөрөгдсөн үйлдлүүдийн аль нэгийг хийх магадлалыг тооцдог неорал сүлжээ юм.

Доорх зурагт үзүүлсэн Понг тоглоомд полиси нь дэлгэцийн пикселийг авч тоглогчийн цохиурыг (ногоон) дээш, доош эсвэл хоёуланг нь хөдөлгөх магадлалыг тооцоолж болно. Доорх зурагт полиси нь неорал сүлжээ буюу Hidden layer юм. [6]



Зураг 1.2: Неороны сүлжээний жишээ

1.1.1 Бататгасан сургалттай холбоотай нэр томъёо

- Action (A): Агент-ийн хийх боломжтой бүх алхмууд
- State (S): Орчноос буцаж ирэх тухайн нөхцөл байдал
- Reward (R): өмнөх үйлдлийн үр дүнд гарсан ололт, шагнал
- Policy (π): Агент одоогийн төлөв байдалд үндэслэн дараагийн үйлдлийг тодорхойлоход ашигладаг стратеги.
- Value (V): урт хугацааны ололт
- Q-value эсвэл action-value(Q): Value-тай төстэй. Гэхдээ одоогийн үйлдлийг нэмэлт параметрээр авдаг.

1.1.2 Model-free

Model-free гэдэг нь мэдлэгээ шинэчлэхийн тулд шагналд тулгуурладаг. Төлөвүүд болон үйлдлүүдийг хадгалах шаардлагагүй.

1.1.3 On-policy болон off-policy

On-policy агент нь value-г одоогийн policy-г ашигласан одоогийн үйлдэлд тулгуурлан сурдаг. Харин off-policy агент өөр нэг policy-г ашигласан үйлдэл a*-д тулгуурлан сурдаг.

1.1.4 Суралцах үйл явц

Reinforcement learning буюу бататгасан сургалтад тасралттай үйлдлийн хувьд суралцах үйл явц нь санамсаргүй үйлдлийг сонгох замаар явагддаг. Харин үргэлжилсэн үйлдлийн хувьд суралцах үйл явц нь үйлдэлд шуугианыг нэмэх замаар явагддаг.

1.2 Гүн неороны сүлжээ

Хиймэл неорал сүлжээ (Artificial Neural Network - ANN) нь тархи хэрхэн ажилладгаас санаа авсан функцийг ойролцоологч (function approximator) юм. ANN нь нэг давхарга дахь неоралууд бүгд урд давхарга дахь неоралуудтай холбогддог хэд хэдэн давхарласан хиймэл неоралаас бүрдэнэ. Бүх хиймэл неоралууд нь идэвхжүүлэх функцтэй байдаг бөгөөд энэ нь ихэвчлэн ReLu функц байдаг.

$$f(x) = max(0, x)$$

X_j нь неоралтай холбогдсон тохиолдолд w_{ij} жинтэй байдаг ба неорал бүр хэвийх утгатай b_i байдаг. Неоралын гаралтыг дараах томъёогоор тооцоолж болно:

$$y = f(b_i + \sum_{h=1}^{n} w_{ij}x_j)$$

Эхний давхаргыг оролт болгон ашигласнаар оролтыг өөр давхаргаар дамжуулж, сүлжээний сүүлийн давхаргаас гарах утгыг авах боломжтой. ANN нь илүү нарийн төвөгтэй функцуудийг

ойролцоолох боломжтой тохиромжтой олон давхарга, неоралуудтай бол гүн неорал сүлжээ (Deep Neural network - DNN) гэж үздэг.

1.3 DDPG алгоритм

Deep Deterministic Policy Gradient (DDPG) бол үргэлжилсэн, тасралтгүй үйлдлүүдийг сурахад зориулагдсан model-free off-policy алгоритм юм. Q-функц ба policy-ыг зэрэг сурдаг алгоритм юм. Q-функцийг сурахын тулд off-policy өгөгдөл болон Bellman тэгшитгэлийг ашигладаг. Мөн policy-г сурахын тулд Q-функцыг ашигладаг. [3]

DDPG алгоритм дараах 4 неорал сүлжээг ашигладаг:

- θ^{Q}:Actor сүлжээ
- θ^{μ}:Critic сүлжээ
- $\theta^{Q'}$:target Actor сүлжээ
- $\theta^{\mu'}$:target Critic сүлжээ

Actor сүлжээ нь төлөвөөс хамааран үйлдлийг санал болгоно. Төлөвийг оролтоор авч үйлдлийг гаргана. Critic сүлжээ нь төлөвөөс хамаарсан үйлдэл нь сайн эсвэл муу болохын урьдчилан таамагладаг. Төлөв болон үйлдлийг оролтоор авч Q-value-г гаргадаг. Actor сүлжээ нь байнгын суралцаж явдаг бол critic сүлжээ нь аажмаар суралцаж явдаг.

Target network нь суралцсан сүлжээнүүдийг хянаж байдаг эх сүлжээнүүдийнхээ хуулбарууд юм. Эдгээр сүлжээг ашиглан тогтвортой сурах байдлыг сайжруулдаг. [2]

Доор DDPG алгоритмын pseudo-code-ыг харууллаа. Үүнийг 4 хэсэгт задлан тайлбарлаж болно. [1]

- Туршлагаа хадгалах (Experience replay)
- Actor болон critic сүлжээг шинэчлэх

- Target сүлжээг шинэчлэх
- Судалгаа хийх (Exploration)

DDPG алгоритмын pseudo code

θ^Q болон θ^μ жинтэйгээр critic сүлжээ $Q(s_i, a_i|\theta^Q)$ болон actor сүлжээ $\mu(s|\theta^m u)$ -г үүсгэнэ

$\theta^{Q'} \longleftarrow \theta^Q$, $\theta^{\mu'} \longleftarrow \theta^\mu$ жинтэйгээр target сүлжээ $Q^{'}$ болон $\mu^{'}$-г үүсгэнэ

Replay buffer-аа үүсгэнэ

for episode = 1, M do

Анхны төлөв болох s1-г авна

for t = 1, T do

Тухайн policy болон шуугиан дээрээ үндэслэн үйлдлээ сонгоно $a_t = \mu(s_t|\theta^\mu) + N_t$

Үйлдэл a_t-гээ гүйцэтгээд reward r_t болон шинэ төлөв $s_t + 1$-ээ авна

Replay buffer-даа төлөв, үйлдэл, reward, шинэ төлөвөө ($s_t, a_t, r_t, s_t + 1$) хадгалж авна

Replay buffer-аасаа N тооны санамсаргүй утгыг авна

$y_i = r_i + \gamma Q^{'}(s_i + 1, \mu^{'}(s_i + 1|\theta^{\mu^{'}})|\theta^{Q'})$ утгыг онооно

Алдааг багасгаж critic сүлжээг шинэчилнэ: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

Actor policy-г шинэчлэнэ:

$$\bigtriangledown_\theta \mu J(\theta) \approx \frac{1}{N}\sum_i [\bigtriangledown_a Q(s, a|\theta^Q)|_s = s_i, a = \mu(s_i) \bigtriangledown_\theta \mu \mu(s|\theta^\mu)|_s = s_i]$$

Target сүлжээнүүдийг шинэчлэнэ:

$$\theta^{Q'} \longleftarrow \tau\theta^Q + (1 - \tau)\theta^{Q'}$$
$$\theta^{\mu'} \longleftarrow \tau\theta^\mu + (1 - \tau)\theta^{\mu'}$$

end for

end for

Replay buffer

DDPG алгоритм нь replay buffer-ыг туршлагыг цуглуулахад ашигладаг. Цуглуулсан туршлагаа неороны сүлжээний параметрүүдийг шинэчлэхэд ашигладаг. Value болон policy сүлжээг шинэчлэхдээ replay buffer дахь туршлагуудаас санамсаргүй байдлаар цуглуулан ашигладаг.

Яагаад replay buffer-ыг ашиглаж байгаа вэ гэхээр алгоритмд хамааралгүй байдлаар тархсан өгөгдөл хэрэгтэй. Ийм өгөгдлүүдийг replay buffer дахь туршлагуудаас санамсаргүй байдлаар сонгон авах байдлаар цуглуулж болно.

Actor болон Critic сүлжээг шинэчлэх

Critic сүлжээг шинэчлэх үйл явц нь Q-learning-тэй төстэй байдлаар хийгддэг. Шинэчлэгдсэн Q утгыг Беллманы тэгшитгэлээс гарган авна:

$$y_i = r_i + \gamma Q^{'}(s_i + 1, \mu^{'}(s_i + 1|\theta^{\mu^{'}})|\theta^{Q'})$$

DDPG-д дараагийн төлөв Q утгуудыг target critic network, target actor network ашиглан тооцдог. Дараа нь шинэчлэгдсэн Q утга ба анхны Q утга хоорондын дундаж квадрат алдааг хамгийн бага хэмжээнд хүртэл бууруулна:

$$Loss = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

Анхны Q утга нь target network-ооc биш critic network-ооc бодогдон гарна.

Actor сүлжээний хувьд гол зорилго нь буцан ирэх үр дүн хамгийн дээд хэмжээнд байх юм:

$$J(\theta) = E[Q(s, a)|_{s=s_t, a_t=\mu(s_t)}]$$

Actor алдагдлыг тооцоолохын тулд зорилгын функцийн деривативыг авна:

$$\bigtriangledown_\theta \mu J(\theta) \approx \bigtriangledown_a Q(s, a)\ \bigtriangledown_\theta\ \mu\mu(s|\theta^\mu)$$

Policy-гоо off-policy байдлаар шинэчилж байгаа учир санамсаргүй байдлаар авсан туршлагуудынхаа градиентүүдийн нийлбэрийн дундаж утгыг авна:

$$\bigtriangledown_\theta \mu J(\theta) \approx \frac{1}{N}\sum_i [\bigtriangledown_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)}\ \bigtriangledown_\theta\ \mu\mu(s|\theta^\mu)|_{s=s_i}]$$

Target сүлжээг шинэчлэх

Target сүлжээний параметрүүдийг хуулбарлаад, тэдгээрээр дамжуулан сурсан сүлжээнүүдээ хянана. Target сүлжээний параметрүүдийг тодорхой хугацааны алхам хийсний дараа дараах томъёогоор шинэчилдэг:

$$\theta^{Q'} \longleftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \longleftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

τ бол ихэвчлэн 1-тэй ойролцоо байхаар сонгосон параметр юм (жишээлбэл: 0.999).

Шуугиан нэмэх

DDPG алгоритмын баримт бичиг зохиогчид үйлдэлд шуугиан нэмэхийн тулд N:Ornstein-Uhlenbeck Process-г ашигласан байна:

$$\mu^{'}(s_t) = \mu(s_t|\theta_t^{\mu}) + N$$

Ornstein-Uhlenbeck процесс нь өмнөх шуугиантай уялдаатай холбоотой шуугианыг бий болгодог.

1.4 Хиперпараметрүүдийн тохируулга

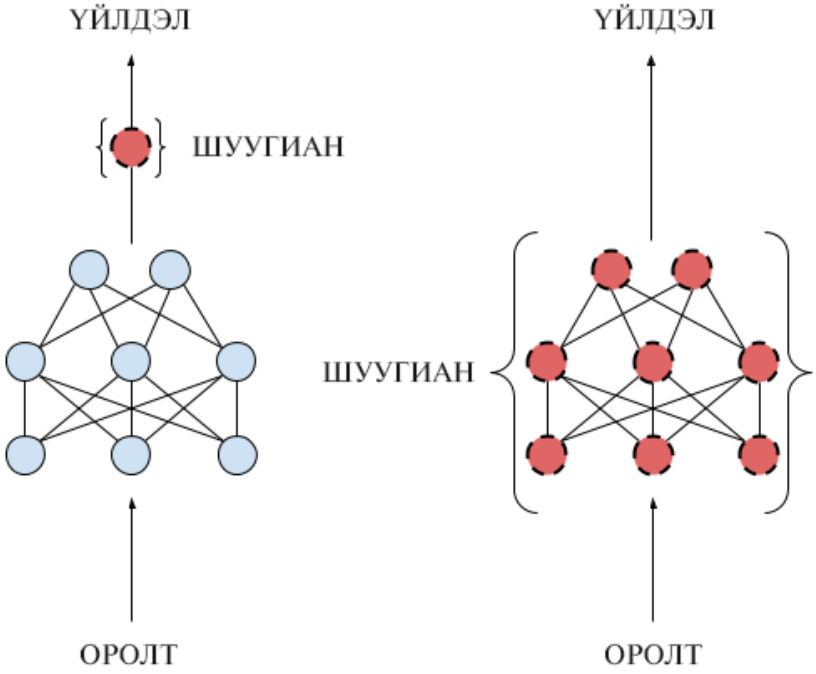
RL алгоритмууд нь алгоритм хэрхэн ажиллахыг өөрчилдөг параметрүүд болох хиперпараметрүүдтэй байдаг. Алгоритм нь тодорхой орчинд сайн ажиллаж байгаа эсэхийг баталгаажуулахын тулд хиперпараметрын өөр өөр утгыг ашиглан туршиж аль хиперпараметрүүд нь хамгийн сайн ажиллаж байгааг болохыг олж мэдэхийн тулд хиперпараметрын тохируулга хийх шаардлагатай байдаг.

1.5 Орчны шуугиан

RL алгоритмд ашигладаг ихэнх полиси нь стохастик буюу санамсаргүй байдлаар тодорхойлогдсон байдаг. Энэ нь зөвхөн аливаа үйлдэл хийх магадлалыг л тооцоолно гэсэн үг. Сургалтын

явцад агент нь тодорхой нэг төлөв байдалд хэд хэдэн удаа орж болох бөгөөд тухайн төлөв бүрт түүвэрлэлтийн (sampling) улмаас өөр өөр үйлдэл хийж болно гэсэн үг юм. Эдгээр үйлдлүүдийн зарим нь оновчтой зарим нь оновчгүй байх бөгөөд оновчгүй үйлдлийг багасгахын тулд агентын үйлдэлд шуугианыг нэмж байгаа. Ерөнхийдөө орчны шуугианыг агент нэг алхмаас нөгөө алхамд хийх үйлдэл бүртэй холбоотой магадлалыг өөрчлөхөд ашиглаж байна гэсэн үг.

Бид үйлдлийн орчны шуугиан (Action space noise), параметр шуугиан (Parameter noise) гэх 2 төрлийн шуугианыг авч үзэх болно. Доорх зургийн зүүн талынх нь үйлдлийн орчны шуугиан, баруун талынх нь параметр шуугиан юм.



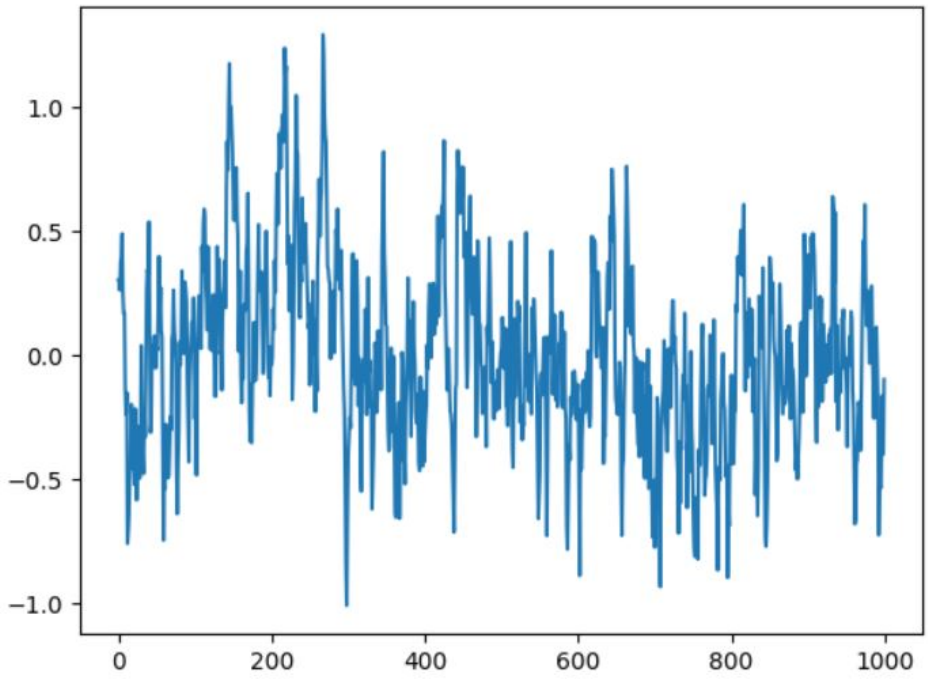
Зураг 1.3: 2 төрлийг орчны шуугиан

1.5.1 Үйлдлийн орчны шуугиан

Үйлдлийн орчны шуугиан гэдэг нь орчны шуугианыг үйлдэл дээр нэмэхийг хэлдэг. Үйлдлийн орчны шуугиан дотор бие биетэйгээ хамааралтай шуугиан, бие биенээсээ хамааралгүй шуугиан гэх 2 шуугианыг авч үзнэ.

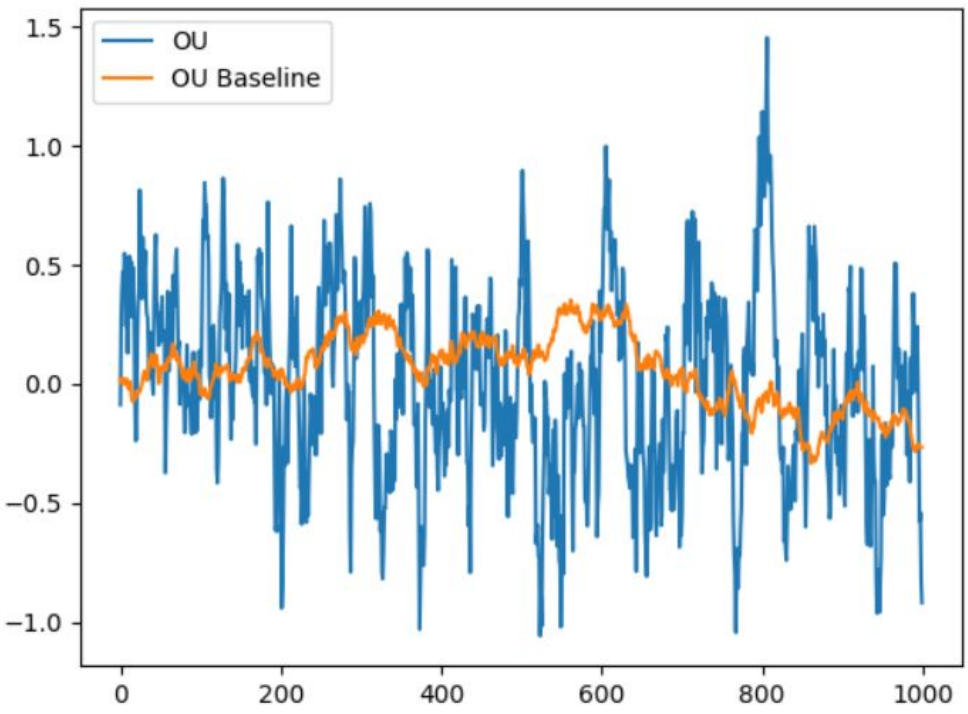
Бие биетэйгээ хамааралтай шуугиан

Correlated noise буюу бие биенээсээ хамааралтай шуугианыг үүсгэхдээ Ornstein–Uhlenbeck-ын процессыг ашиглаж байгаа. Энэ процесс нь бие биетэйгээ хамааралтай шуугианыг үүсгэж өгч байгаа. Доорх графикт ямархуу тархац, савалгаатай шуугиан үүсгэж байгаа харууллаа. Хэвтээ тэнхлэгийн дагуу хэдэн удаа үүсгэсэн, босоо тэнхлэгийн дагуу ямар утга авах нь харгалзаж байна:



Зураг 1.4: OU процессын график

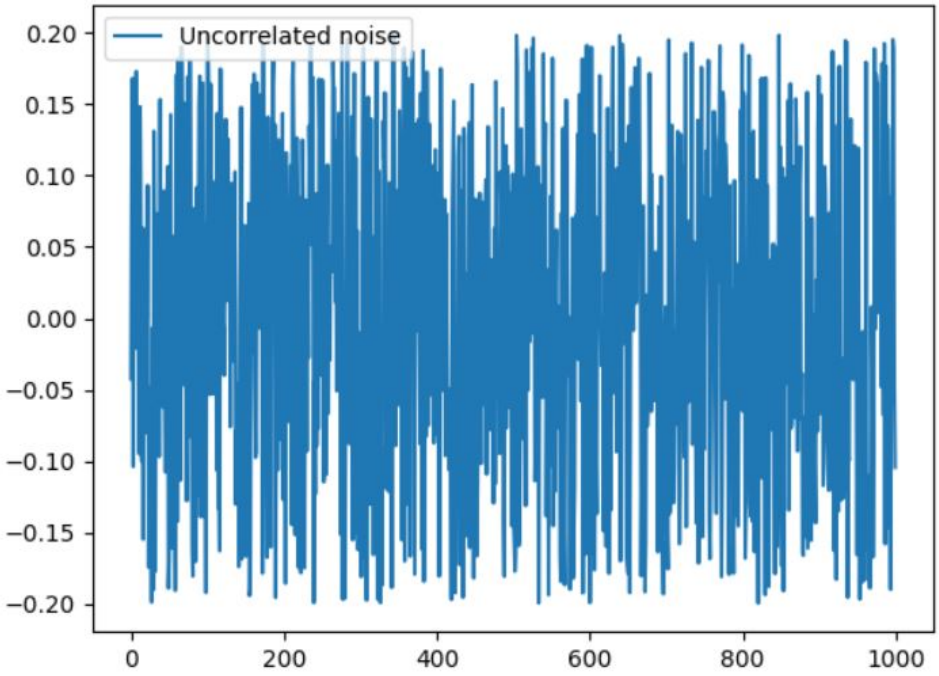
Туршилт хийхдээ хоёр өөр утга гаргах Ornstein–Uhlenbeck-ын процессын хэрэгжүүлэлтийг ашигласан. Нэг нь илүү тогтвортой алсуур аа савладаг, нөгөө тогтворгүй савлалт ихтэй байна. Доорх графикт ялгааг харууллаа:



Зураг 1.5: 2 өөр OU процессын график

Бие биенээсээ хамааралгүй шуугиан

Uncorrelated noise буюу бие биенээсээ хамааралгүй шуугианыг үүсгэхдээ [-0.2, 0.2]-ын хооронд санамсаргүй тоо авах байдлаар үүсгэж байгаа. Яагаад [-0.2, 0.2]-ын хооронд авч байгаа вэ гэхээр 3 төрлийн шуугианаа хооронд нь харьцуулах учраас адилхан параметртэй хэрэгжүүлж байгаа. Доорх графикт ямархуу тархац, савалгаатай шуугиан үүсгэж байгаа харууллаа (1000 удаа үүсгэсэн):



Зураг 1.6: Бие биенээсээ хамааралгүй шуугианы график

1.5.2 Параметр шуугиан

Параметр шуугиан нь бусад аргуудаас илүү үр дүнтэй арга юм. Параметр шуугиан нь үйлдэлд шуугиан нэмэх бус неорал сүлжээн дээр параметр байдлаар шуугианыг нэмдэг. Шуугиан нь дасан зохицдог шуугиан (Adaptive noise) байна. Уламжлалт RL нь үйлдлийн орчны шуугианыг ашиглан агент нэг алхмаас нөгөө алхамд хийх үйлдэл бүртэй холбоотой магадлалыг өөрчилдөг. Параметр шуугиан нь алгоритмыг орчноо илүү үр дүнтэй судалж, илүү өндөр оноо авахад тусалдаг. [7]

1.5.3 Параметр шуугианыг хэрэгжүүлэх

Episode болгоны эхэнд actor сүлжээг хуулж аваад Гауссын шуугианыг нэмж шинэ шуугиантай actor сүлжээ үүсгэнэ.

$$\tilde{\theta}^Q = \theta^Q + N(0, \sigma^2)$$

Тухайн episode шинэ үүссэн actor сүлжээн дээр сургагдана. Episode дууссаны дараа шинэ үүссэн actor сүлжээний үйлдлүүд болон энгийн actor сүлжээний үйлдлүүд хоорондын зай тооцно. Зайг тооцохдоо:

$$d(\theta^Q, \tilde{\theta^Q}) = \sqrt{1/N \sum_{i=1}^{n} E_s[(\theta^Q(s)_i - \theta^Q\tilde{(s)}_i)^2]}),$$

N = үйлдлийн хэмжээс

Шинэ үүссэн actor сүлжээний үйлдлүүд болон энгийн actor сүлжээний үйлдлүүд хоорондын зөрүүний квадратуудын нийлбэрийг олоод үүний язгуурыг олж байна.

Энэ нь ямар байгаагаас хамааран параметр шуугианыхаа сигмаг өөрчилнө:

$$\sigma_{k+1} = \begin{cases} \alpha\sigma_k & \text{Хэрвээ } d(\theta^Q, \tilde{\theta^Q}) \leq \delta, \\ \frac{1}{\alpha}\sigma_k & \text{Бусад тохиолдолд ,} \end{cases}$$

Дээрх байдлаар шуугианыхаа σ-г өөрчилөн явна. Ийм байдлаар хэрэгжүүлэлтийг хийх болно.

Off-policy аргад зориулсан параметер шуугиан

Off-policy үед суралцах үйл явцыг (exploration) шинэ үүсгэсэн actor сүлжээн дээр хийх бөгөөд сургах процессоо энгийн actor сүлжээн дээрээ хийнэ.

1.6 Ашигласан технологи

1.6.1 Gym

Gym бол reinforcement learning буюу бататгасан сургалтын алгоритмуудыг хөгжүүлэх болон харьцуулахад зориулагдсан хэрэгсэл юм. Үүнийг ашиглан агентдаа алхах, тоглоом тоглох зэрэг бүх зүйлийг зааж болно.

Яагаад үүнийг ашигладаг вэ?

Бататгасан сургалт (RL) нь шийдвэр гаргахтай холбоотой машин сургалтын дэд талбар юм. Энэ нь агент нарийн төвөгтэй, тодорхойгүй орчинд хэрхэн зорилгодоо хүрч болохыг судалдаг. RL нь доорх 2 шалтгааны улмаас ихээр ашиглагдаж байна:

- RL нь дараалсан шийдвэр гаргахтай холбоотой бүхий л асуудлыг багтаасан байдаг. Жишээ нь роботын хөдөлгүүрийг удирдаж түүнийг үсрэх чадвартай болгох, үнэ, бараа материалын менежмент гэх мэт бизнесийн шийдвэр гаргах, видео тоглоом, ширээний тоглоом тоглох гэх мэт
- RL алгоритмууд олон хүнд хэцүү орчинд сайн үр дүнд хүрч эхэлсэн

Гэсэн хэдий ч RL судалгааны ажлыг RL-ын open-source орчин хангалттай олон янз байдаггүй бөгөөд тэдгээрийг тохируулах, ашиглахад хэцүү байдал болон орчны стандартчилал дутмаг гэсэн хоёр хүчин зүйл удаашруулж байна. Gym нь эдгээр 2 асуудлыг шийдэхийг зоридог.

1.6.2 BiPedalWalker-v2

Энэ бол gym-ын Box2D симулятор дахь нэг орчин юм. Гол зорилго нь bipedal роботыг алхаж сургах. Урагш алхах бүрд reward өгдөг. Нийтдээ төгсгөл хүртэл 300+ оноог өгдөг. Хэрэв робот унавал -100 оноо өгдөг. Илүү сайн агент нь илүү сайн оноо авах болно.

Төлөв нь их биений өнцгийн хурд, өнцгийн хурд, хэвтээ хурд, босоо хурд, хөлний байрлал, хөлний өнцгийн хурд, хөлтэй газар шүргэлцэх, 10 лидарын зай хэмжигч хэмжигдэхүүнээс бүрдэнэ. Мужийн векторт координат байхгүй байна.

1.6.3 Pytorch

Pytorch бол Torch сан дээр тулгуурласан open-source машин сургалтын сан юм. Python хэлэнд ихээр ашиглагддаг ч мөн C++ програмчлалын хэлд ашиглагддаг. Энэ нь GPU ашигладаг. PyTorch нь өндөр түвшний хоёр онцлог шинж чанарыг агуулдаг:

- GPU-г ашиглан тензорын тооцоолол хийх (NumPy гэх мэт)
- Гүнзгий неороны сүлжээг (Deep neural network) бий болгох

2016 оны 1 сард гарсан бөгөөд үүнээс хойш олон судлаачид үүнийг ашигласаар байна. Учир нь маш нарийн төвөгтэй неороны сүлжээг хялбараар бий болгодог. Мөн кодоо шалгахдаа заавал бүхлээр нь ажиллуулах шаардлагагүй болсон. Шаардлагатай тохиолдолд Pytorch-ын функцуудыг NumPy, SciPy, Cython зэргээр өргөтгөж болно.

2. ХЭРЭГЖҮҮЛЭЛТ

2.1 Алгоритмын хэрэгжүүлэлт

Хэрэгжүүлэлтийг python програмчлалын хэлийг ашиглан гүйцэтгэсэн. Орчинг бэлдэхдээ gym openai хэрэгслийг ашигласан. Pytorch санг тооцоолол хийх, неороны сүлжээ үүсгэх зэрэгт ашигласан. Дээр дурдсан DDPG алгоритмын pseudo кодын дагуу кодыг бичсэн. 3 шуугианы гүйцэтгэлийг харьцуулах учраас ойролцоо хипер параметрүүдийг ашигласан.

DDPG алгоритмын хувьд Lillicrap et al.(2015)[4]-дээр тайлбарласантай төстэй actor, critic сүлжээний архитектурыг ашигласан. Target сүлжээнүүдийг $\tau = 0.001$ гэсэн параметртэй soft-update хийсэн. Actor, critic сүлжээ хоёулаа Adam сургагчийг ашигласан.

Кодыг хавсралт хэсэг оруулсан. Github-аас харахыг хүсвэл дараах холбоосоор хандана уу `https://github.com/bbyambanyam/ddpg_algorithm.git`

2.2 Алгоритмын турших орчин

BipedalWalker-v3 орчин дээр алгоритмыг ажиллуулан туршилаа.

Зураг 2.1: BiPedalWalker-V3 орчин

Орчинг үүсгэж төлөвийн хэмжээс, үйлдлийн хэмжээс, хийж болох үйлдлийн тоог авна.

```
env = gym.make('BipedalWalker-v3')


state_dimension = env.observation_space.shape[0]
action_dimension = env.action_space.shape[0]
action_max = env.action_space.high[0]
```

Код 2.1: Орчин үүсгэх

2.2.1 Хипер параметрүүд

Доорх DDPG-ийн хипер параметрүүд нь BiPedalWalker-V3 орчныг шийдвэрлэхэд тохируулагдсан болно. Зарим тохиолдолд өөр байж болно.

- Actor learning rate: 0.0001 (Adam optimizer)

- Critic learning rate: 0.001 (Adam optimizer)
- Memory buffer size: 1000000
- Minibatch size: 128
- OU-noise-theta: 0.15
- OU-noise-sigma: 0.2
- OU-noise-mu: 0
- normal-noise: 0.2
- Steps: 1600
- Target update: 0.001
- initial-stddev (σ): 0.1
- desired-action-stddev (δ): 0.2
- adaptation-coefficient(α): 1.01

2.3 Чухал кодын хэсгүүд

Actor, target actor, critic болон target critic сүлжээг optimizer (сургагч)-ын хамт үүсгэх. Үүсгэхдээ төлөвийн хэмжээс, үйлдлийн хэмжээс, хийж болох үйлдлийн тоо зэргийг ашиглана.

```
actor = models.Actor(state_dimension, action_dimension, action_max)
target_actor = models.Actor(state_dimension, action_dimension,
   action_max)
actor_optimizer = torch.optim.Adam(actor.parameters(), lr=0.001)

critic = models.Critic(state_dimension, action_dimension)
```

```
target_critic = models.Critic(state_dimension, action_dimension)
critic_optimizer = torch.optim.Adam(critic.parameters(), lr=0.001)

for target_param, param in zip(target_actor.parameters(), actor.
    parameters()):
  target_param.data.copy_(param.data)

for target_param, param in zip(target_critic.parameters(), critic.
    parameters()):
    target_param.data.copy_(param.data)
```

Код 2.2: Actor critic сүлжээ үүсгэх

Replay buffer үүсгэх. Buffer-аа үүсгэснийхээ дараа hot start хийхийн тулд санамсаргүй утгуудаар дүүргэж өгсөн.

```
ram = memory.ReplayBuffer(1000000))
```

Код 2.3: Replay buffer үүсгэх

Ornstein-Uhlenbeck Process-г үүсгэх

```
noise = utilities.OrnsteinUhlenbeckActionNoise(action_dimension)
```

Код 2.4: Шуугиан үүсгэх

Дээр үүсгэсэн шуугианаа ашиглан үйлдэл дээр correleted шуугианыг нэмэх

```
action_with_noise = action_without_noise.data.numpy() + (noise.sample()
    * action_max)
```

Код 2.5: Үйлдэл дээр шуугиан нэмэх

Үйлдэл дээр uncorreleted шуугианыг нэмэх

```
action_with_noise = action_without_noise.data.numpy() + (random.uniform
    (-0.2, 0.2) * action_max)
```

Код 2.6: Үйлдэл дээр шуугиан нэмэх

Параметр шуугиан үүсгэх

```
#Parameter noise-d zoriulsan actor
actor_copy = models.Actor(state_dimension, action_dimension, action_max
    )

#Parameter noise
parameter_noise = utilities.AdaptiveParamNoiseSpec(initial_stddev=0.05,
    desired_action_stddev=0.3, adaptation_coefficient=1.05)
```

Код 2.7: Үйлдэл дээр шуугиан нэмэх

Параметр шуугианд зориулан үүсгэсэн шинэ actor сүлжээний неорал сүлжээн дээр параметр шуугианыг нэмэх

```
parameters = actor_copy.state_dict()
for name in parameters:
    parameter = parameters[name]
    rand_number = torch.randn(parameter.shape)
    parameter = parameter + rand_number * parameter_noise.
        current_stddev
```

Код 2.8: Үйлдэл хийх

Үйлдлийг хийж шинэ төлөв, reward-г авах

```
new_observation, reward, done, info = env.step(action_with_noise)
```

Код 2.9: Үйлдэл хийх

Параметр шуугианы зай тооцох, сигмаг шинэчлэх

```
#Distance tootsoh
diff_actions = actor_copy_actions - actor_actions
mean_diff_actions = np.mean(np.square(diff_actions),axis=0)
distance = math.sqrt(np.mean(mean_diff_actions))

#Sigma-g update hiih
parameter_noise.adapt(distance)
```

Код 2.10: Үйлдэл хийх

Critic сүлжээг сургаад, шинэчлэх

```
# Critic network-g surgah

predicted_action = target_actor.forward(next_states).detach()
next_val = torch.squeeze(target_critic.forward(next_states,
   predicted_action).detach())
y_expected = rewards + 0.99*next_val
y_predicted = torch.squeeze(critic.forward(states, actions))

# Critic network-g shinechleh, critic loss-g tootsooloh

critic_loss = F.smooth_l1_loss(y_predicted, y_expected)
critic_optimizer.zero_grad()
critic_loss.backward()
critic_optimizer.step()
```

Код 2.11: Critic сүлжээг сургах шинэчлэх

Actor сүлжээг сургах

```
predicted_action = actor.forward(states)
actor_loss = -1*torch.sum(critic.forward(states, predicted_action))
actor_optimizer.zero_grad()
actor_loss.backward()
actor_optimizer.step()
```

Код 2.12: Actor сүлжээг сургах

3. ҮР ДҮНГИЙН БОЛОВСРУУЛАЛТ

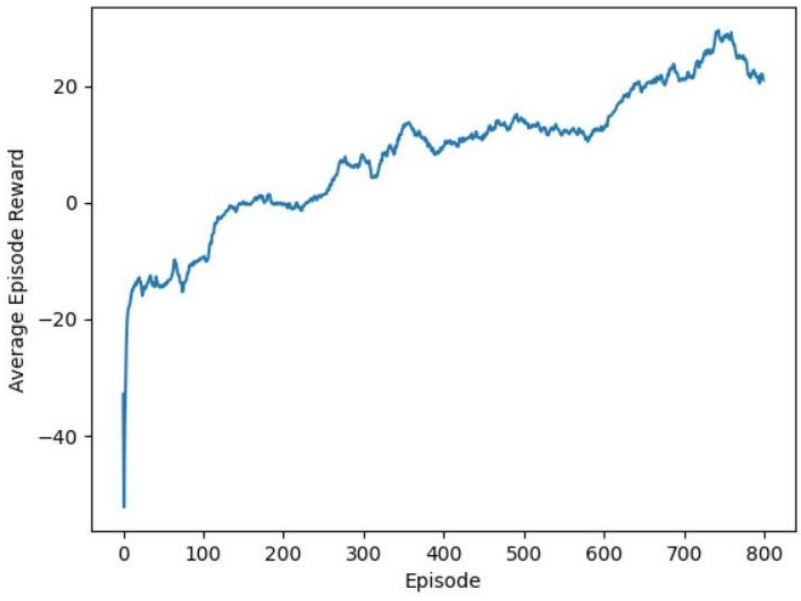
3.1 Туршилтын үр дүн

Хэрэв алгоритм зөв ажиллаж байгаа бол reward нь өсөж байх ёстой байдаг. Дундаж reward-ыг авахдаа episode болгоны нийлбэр reward-г олоод үүнийгээ list-д хадгалан аваад энэ list-ээс сүүлийн 40 үр дүнгийн дундажыг олж графикийг зурсан.

3.1.1 Үйлдлийн орчны шуугиан

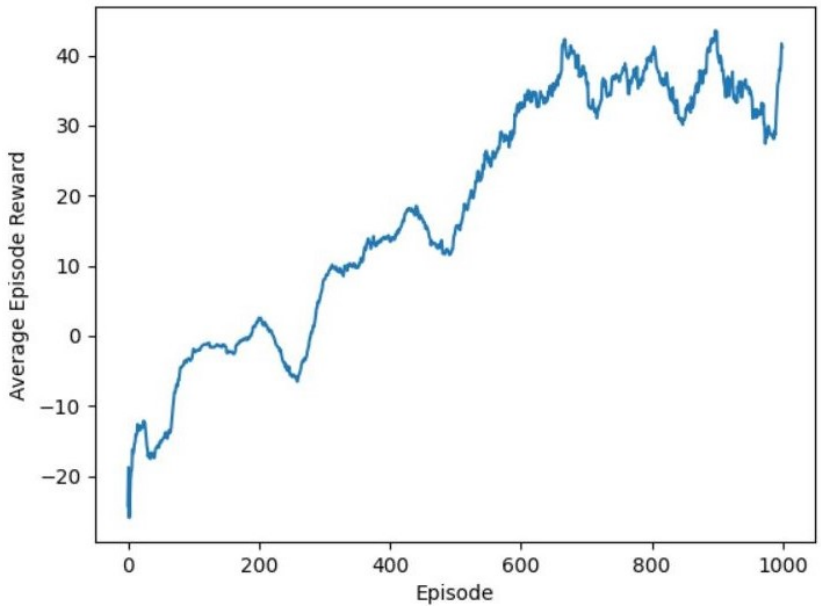
Correlated шуугиан

Доорх графикийн хувьд 800 episode ажилласан бөгөөд босоо тэнхлэгийн дагуу дундаж reward, хэвтээ тэнхлэгийн дагуу ажилласан episode-г авч байна. Орчны шуугианыг нэмэхдээ Ornstein-Uhlenbeck Process-г ашигласан. Доорх бүх графикийн хувьд Actor LEARNING RATE: 0.001 Critic LEARNING RATE: 0.001 бусад хипер параметрүүд ижил.



Зураг 3.1: Correlated noise

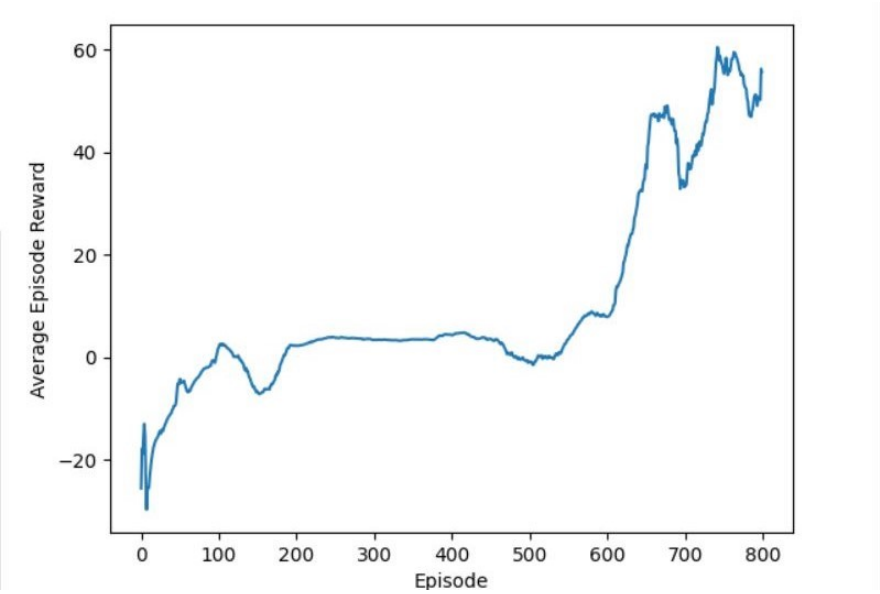
1000 episode ажиллуулсны дараа:



Зураг 3.2: Correlated noise

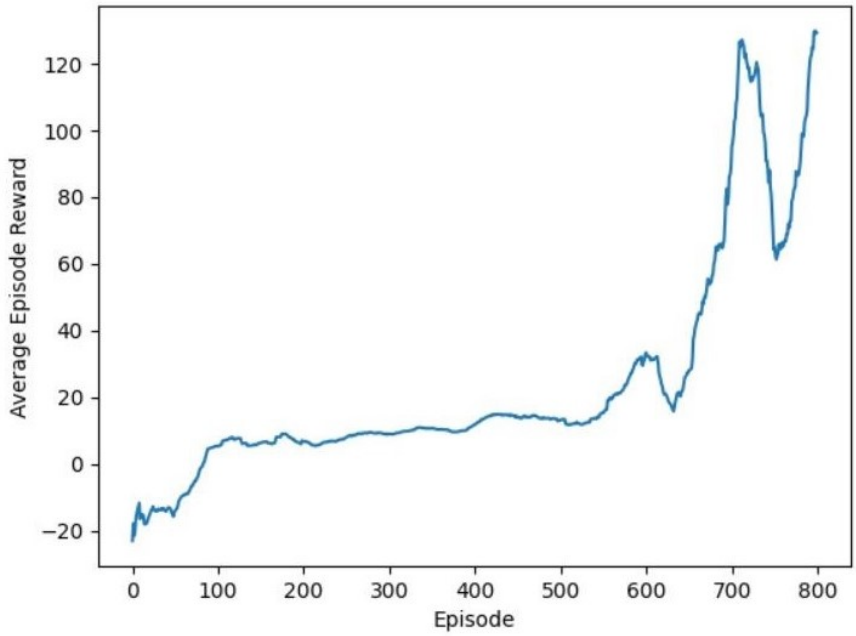
Uncorrelated шуугиан

Доорх хоёр графикийн хувьд 800 episode ажилласан бөгөөд орчны шуугианыг -0.2 оос 0.2-ын хооронд санамсаргүй байдлаар сонгон авч үйлдэл дээрээ нэмж байгаа. Энэ нь өмнөх шуугиантай уялдаа хамааралгүй буюу uncorrelated шуугиан гэсэн үг юм. Доорх бүх грапикийн хувьд Actor LEARNING RATE: 0.001 Critic LEARNING RATE: 0.001 бусад хипер параметрүүд ижил.



Зураг 3.3: Uncorrelated noise

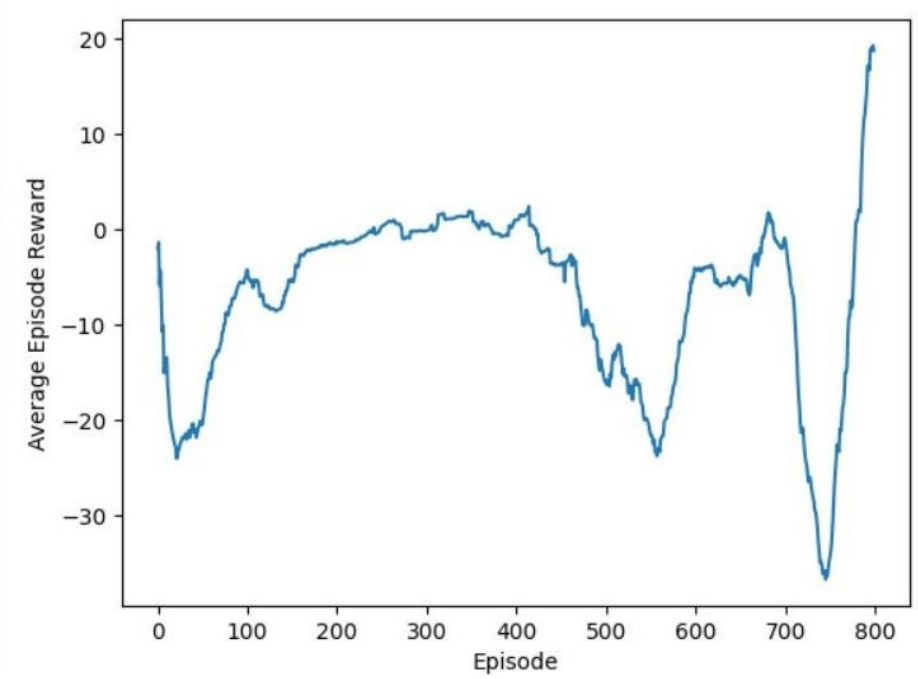
Uncorreleted шуугианы өөр нэг график:



Зураг 3.4: Uncorrelated noise 2

Доорх графикийн хувьд 800 episode ажилласан бөгөөд орчны шуугианыг -0.4-оос 0.4-

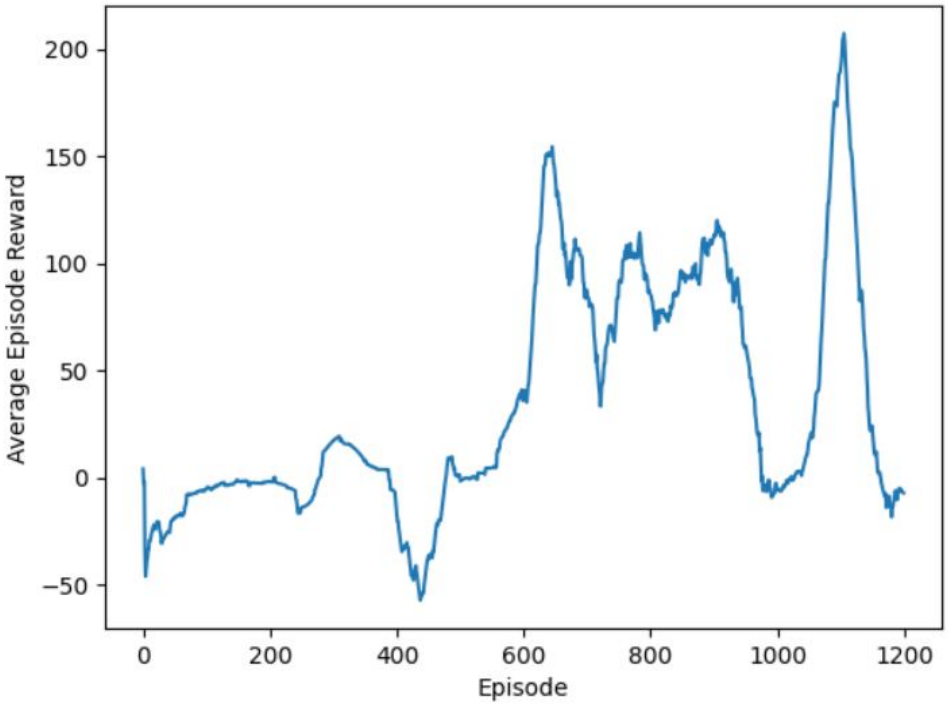
ын хооронд санамсаргүй байдлаар сонгон авч үйлдэл дээрээ нэмж байгаа. Энэ нь мөн адил өмнөх шуугиантай уялдаа хамааралгүй буюу uncorrelated шуугиан юм. Графикаас харахад 0.4 өөр сонгон авсан үед нь үр дүн муу гарч байна.



Зураг 3.5: Uncorrelated noise 0.4

3.1.2 Parameter Space Noise

Доорх параметр шуугианы үр дүнг нь гаргахын тулд 1200 episode ажиллуулсан. ACTOR LEARNING RATE: 0.0001, CRITIC LEARNING RATE: 0.001:

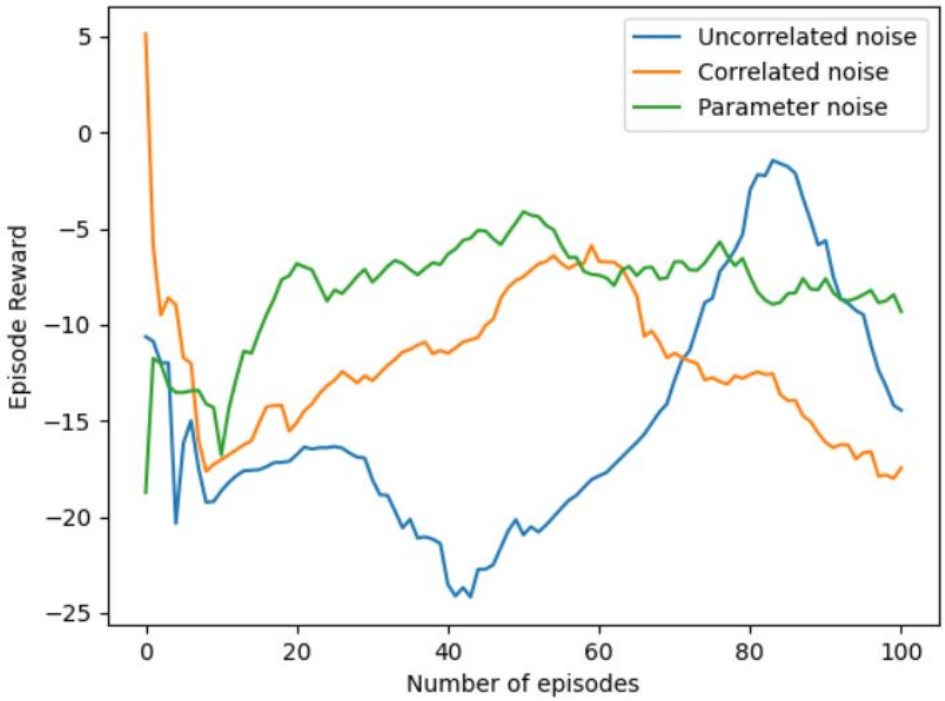


Зураг 3.6: Параметр орчны шуугиан

3.2 Үр дүнгийн харьцуулалт

Харьцуулалт хийхийн тулд бүх шуугиан дээр ижил хипер параметртэйгээр ажиллуулсан. Uncorrelated noise буюу бие биенээсээ хамааралгүй шуугиан ($\theta = 0, \sigma = 0.2$), Correlated noise буюу бие биетэйгээ хамааралтай шуугиан (Ornstein–Uhlenbeck process), Parameter space noise буюу параметр шуугиан гэсэн гурван шуугианы харьцуулалтыг хийлээ. Мөн үр дүнг хооронд оо харьцуулагдах боломжтой болгохын тулд параметр шуугианыг дангаар нь ашиглахгүйгээр Ornstein-Uhlenbeck-ын процесстой цуг ашигласан.

Доорх графикийн хувьд үр дүнг гаргахдаа 101 episode ажиллуулсан бөгөөд нэг episode-д хийх алхам буюу MAX-STEP нь 10000 байна. Гүйцэтгэлийг үнэлэхийн тулд эхний 20 episode-ыг ямар ч шуугиангүйгээр ажиллуулсан.



Зураг 3.7: Харьцуулалтын график

Графикаас харахад бие биенээсээ хамааралгүй шуугианы хувьд савалгаа ихтэй байгаа нь харагдаж байна. Дундаж шагнал нь episode 80 дээр их хэмжээтэй гарч байгаа ч савалгаа их тул буурах шинжтэй байна. Бие биетэйгээ хамааралтай шуугианы хувьд савалгаа дундаж байгаа ч дундаж шагналын хувьд харьцангуй бага байна. Параметр шуугианы хувьд савалгаа маш бага дундаж шагнал нь бага байна. Үүнээс дүгнэхэд гүйцэтгэлийн хувьд параметр шуугиан нь маш бага зэрэг илүү байна. Мөн үүнийг батлах өөр нэг үр дүн бол Зураг 3.6 бөгөөд уг зурагт дундаж шагналын хамгийн дээд хэмжээ нь 200+ байна. Бусад шуугианы хувьд дундаж шагнал ийм хэмжээнд хүрээгүй. BipedalWalker-V3 орчин дээр роботын алхах чадварын хувьд параметр шуугианы хувьд илүү байсан. Гэхдээ параметр шуугианы хувьд илт давуу сайн биш байгаа бөгөөд хүлээлтэд хүрээгүй үр дүн гарсан нь хэрэгжүүлэлтийн үед алдаа гарсан байж болзошгүй эсвэл хипер параметрүүд сайн тараагүй гэж харж байна.

Дүгнэлт

Энэхүү судалгааны ажлаар 3 төрлийн шуугианыг DDPG алгоритмыг ашиглан BipedalWalker-V3 орчин дээр амжилттай туршиж гүйцэтгэлээ. Бие биенээсээ хамааралгүй болон бие биетэйгээ хамааралтай шуугианы хувьд үр дүнг хангалттай сайн гарсан гэж харж байгаа бол параметр шуугианы хувьд үр дүн хангалттай сайн сайн биш ч бүр муу үр дүн гараагүй гэж харж байна. Үр дүнгээс үзэхэд шуугианыг нэмэх хамгийн сайн арга бол үйлдэлд шууд шуугиан нэмэх бус неорал сүлжээнд параметр байдлаар шуугианыг нэмэх нь илүү гэдгийг харж болохоор байна. Роботын алхах чадвараас харахад параметр шуугиан нь орчны асуудлыг илүү сайн шийдвэрлэж чадаж байгааг харж болохоор байсан.

Хэрэгжүүлэлтийн явцад гаргасан нэг алдаа бол сургасан моделиудаа хадгалдаг, хадгалсан моделио уншиж аваад үргэлжлүүлэн сургадаг үйлдлийг параметр шуугианыг хэрэгжүүлснийхээ дараа хийх бус анх DDPG алгоритмаа хэрэгжүүлэхдээ цуг хийх байсан юм. Үүнийг эрт хэрэгжүүлээгүйгээс болж бие биенээсээ хамааралгүй шуугиан болон бие биетэйгээ хамааралтай шуугианы моделийг хадгалан аваагүй учир харьцуулалт хийхэд энэ хоёр шуугианыг дахин сургах асуудал гарсан.

Bibliography

[1] Deep Deterministic Policy Gradients Explained, TowardsDataScience, https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b

[2] Deep Deterministic Policy Gradient (DDPG), Keras, https://keras.io/examples/rl/ddpg_pendulum/

[3] Deep Deterministic Policy Gradient, Spinning Up, https://spinningup.openai.com/en/latest/algorithms/ddpg.html

[4] Continuous Control With Deep Reinforcement Learning, Lillicrap et al 2015, https://arxiv.org/pdf/1509.02971.pdf

[5] Parameter noise, Openai, https://openai.com/blog/better-exploration-with-parameter-noise

[6] Evolution stratagies, Openai, https://openai.com/blog/evolution-strategies/

[7] Parameter space noise for exploration, Arxiv, https://arxiv.org/abs/1706.01905

[8] Benchmarking Deep Reinforcement Learning for Continuous Control, Arxiv, https://arxiv.org/abs/1604.06778

[9] Policy Gradient Algorithms. 2018., LilianWeng, https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

А. КОДЫН ХЭРЭГЖҮҮЛЭЛТ

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

def fanin_init(size, fanin=None):
    fanin = fanin or size[0]
    v = 1. / np.sqrt(fanin)
    return torch.Tensor(size).uniform_(-v, v)

class Actor(nn.Module):
    def __init__(self, state_dimension, action_dimension, action_max):
        super(Actor, self).__init__()

        self.state_dimension = state_dimension
        self.action_dimension = action_dimension
        self.action_max = action_max

        # 3 layer uusgene

        self.fc1 = nn.Linear(state_dimension, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 64)
        self.fc4 = nn.Linear(64, action_dimension)

        self.fc1.weight.data = fanin_init(self.fc1.weight.data.size())
        self.fc2.weight.data = fanin_init(self.fc2.weight.data.size())
        self.fc3.weight.data = fanin_init(self.fc3.weight.data.size())
        self.fc4.weight.data.uniform_(-0.003, 0.003)

    def forward(self, state):
        output = self.fc1(state)
        output = F.relu(output)

        output = self.fc2(output)
        output = F.relu(output)

        output = self.fc3(output)
        output = F.relu(output)

        action = self.fc4(output)
        action = F.tanh(action)

        action = action * self.action_max

        # action butsaana

        return action
```

```
49
50 class Critic(nn.Module):
51     def __init__(self, state_dimension, action_dimension):
52         super(Critic, self).__init__()
53
54         self.state_dimension = state_dimension
55         self.action_dimension = action_dimension
56
57         # 4 layer uusgene
58
59         self.fcs1 = nn.Linear(state_dimension, 256)
60         self.fcs2 = nn.Linear(256, 128)
61         self.fca1 = nn.Linear(action_dimension, 128)
62         self.fc2 = nn.Linear(256, 128)
63         self.fc3 = nn.Linear(128, 1)
64
65         self.fcs1.weight.data = fanin_init(self.fcs1.weight.data.size()
               )
66         self.fcs2.weight.data = fanin_init(self.fcs2.weight.data.size()
               )
67         self.fca1.weight.data = fanin_init(self.fca1.weight.data.size()
               )
68         self.fc2.weight.data = fanin_init(self.fc2.weight.data.size())
69         self.fc3.weight.data.uniform_(-0.003, 0.003)
70
71     def forward(self, state, action):
72         s1 = self.fcs1(state)
73         s1 = F.relu(s1)
74         s2 = self.fcs2(s1)
75         s2 = F.relu(s2)
76         a1 = self.fca1(action)
77         a1 = F.relu(a1)
78
79         output = torch.cat((s2, a1), dim=1)
80         output = self.fc2(output)
81         output = F.relu(output)
82         q_value = self.fc3(output)
83
84         # Q value-g butsaana
85
86         return q_value
```

Код А.1: models.py

```
1 import numpy as np
2 from collections import deque
3 import random
4
5 # experience-uudiig hadgalah buffer
6
7 class ReplayBuffer:
8     def __init__(self, size):
```

```
        self.size = size
        self.buffer = deque(maxlen=self.size)

    # experience-g buffer-t nemeh

    def add(self, state, action, reward, next_state):
        exp = (state, action, reward, next_state)
        self.buffer.append(exp)

    # Randor-oor size-toonii experience-g awah

    def sample_exp(self, size):
        batch = []
        size = min(size, len(self.buffer))
        batch = random.sample(self.buffer, size)

        states = np.float32([arr[0] for arr in batch])
        actions = np.float32([arr[1] for arr in batch])
        rewards = np.float32([arr[2] for arr in batch])
        next_states = np.float32([arr[3] for arr in batch])

        return states, actions, rewards, next_states
```

Код А.2: memory.py

```
import numpy as np

#Ornstein-uhlenbeck process-g correlated noise uusgehed ashiglana

#http://math.stackexchange.com/questions/1287634/implementing-ornstein-uhlenbeck-in-matlab

class OrnsteinUhlenbeckActionNoise:

  def __init__(self, action_dim, mu = 0, theta = 0.15, sigma = 0.2):
    self.action_dim = action_dim
    self.mu = mu
    self.theta = theta
    self.sigma = sigma
    self.X = np.ones(self.action_dim) * self.mu

  def reset(self):
    self.X = np.ones(self.action_dim) * self.mu

  def sample(self):
    dx = self.theta * (self.mu - self.X)
    dx = dx + self.sigma * np.random.randn(len(self.X))
    self.X = self.X + dx
    return self.X

class ActionNoise(object):
    def reset(self):
```

```
        pass

#Based on https://github.com/openai/baselines/tree/master/baselines/ddpg
class OrnsteinUhlenbeckActionNoiseBaseline(ActionNoise):
    def __init__(self, mu, sigma, theta=.15, dt=1e-2, x0=None):
        self.theta = theta
        self.mu = mu
        self.sigma = sigma
        self.dt = dt
        self.x0 = x0
        self.reset()

    def __call__(self):
        x = self.x_prev + self.theta * (self.mu - self.x_prev) * self.dt + self.sigma * np.sqrt(self.dt) * np.random.normal(size=self.mu.shape)
        self.x_prev = x
        return x

    def reset(self):
        self.x_prev = self.x0 if self.x0 is not None else np.zeros_like(self.mu)

    def __repr__(self):
        return 'OrnsteinUhlenbeckActionNoise(mu={}, sigma={})'.format(self.mu, self.sigma)

class AdaptiveParamNoiseSpec(object):
    def __init__(self, initial_stddev=0.1, desired_action_stddev=0.2, adaptation_coefficient=1.01):
        self.initial_stddev = initial_stddev
        self.desired_action_stddev = desired_action_stddev
        self.adaptation_coefficient = adaptation_coefficient

        self.current_stddev = initial_stddev

    def adapt(self, distance):
        if distance > self.desired_action_stddev:
            # Decrease stddev.
            self.current_stddev /= self.adaptation_coefficient
        else:
            # Increase stddev.
            self.current_stddev *= self.adaptation_coefficient

    def get_stats(self):
        stats = {
            'param_noise_stddev': self.current_stddev,
        }
        return stats

    def __repr__(self):
```

```
        fmt = 'AdaptiveParamNoiseSpec(initial_stddev={},
            desired_action_stddev={}, adaptation_coefficient={})'
        return fmt.format(self.initial_stddev, self.
            desired_action_stddev, self.adaptation_coefficient)

if __name__ == '__main__':
    ou = OrnsteinUhlenbeckActionNoise(1)
    ou_baseline = OrnsteinUhlenbeckActionNoiseBaseline(mu=np.zeros(1),
        sigma=float(0.2) * np.ones(1))

    states = []
    states2 = []

    for i in range(1000):
        states.append(ou.sample())
        states2.append(ou_baseline())

    import matplotlib.pyplot as plt

    plt.plot(states, label = "OU")
    plt.plot(states2, label = "OU Baseline")
    plt.legend()
    plt.show()
```

Код A.3: utilities.py

```
import gc
import random
import gym
import numpy as np
import matplotlib.pyplot as plt
from torch.autograd import Variable
import torch
import torch.nn.functional as F
import pickle
import os
import math
import time

import memory
import models
import utilities

if __name__ == "__main__":

    ACTOR_LEARNING_RATE = 0.0001
    CRITIC_LEARNING_RATE = 0.001
    MAX_EPISODE = 1201
    MAX_STEPS = 1600
    NOISE_TYPE = "ParameterWithOU" #OU, OUBaseline, Parameter,
        Uncorrelated, NoNoise, ParameterWithOU
    # Orchin beldeh
```

```
    env = gym.make('BipedalWalker-v3')

    state_dimension = env.observation_space.shape[0]
    action_dimension = env.action_space.shape[0]
    action_max = env.action_space.high[0]

    print("State dimension: {}" .format(state_dimension))
    print("Action dimension: {}" .format(action_dimension))
    print("Action max: {}" .format(action_max))

    load_models = False

    # Actor network, critic network uusgeh

    actor = models.Actor(state_dimension, action_dimension, action_max)
    target_actor = models.Actor(state_dimension, action_dimension,
       action_max)
    actor_optimizer = torch.optim.Adam(actor.parameters(), lr=
       ACTOR_LEARNING_RATE)

    critic = models.Critic(state_dimension, action_dimension)
    target_critic = models.Critic(state_dimension, action_dimension)
    critic_optimizer = torch.optim.Adam(critic.parameters(), lr=
       CRITIC_LEARNING_RATE)

    # Target network-g huulah

    for target_param, param in zip(target_actor.parameters(), actor.
       parameters()):
        target_param.data.copy_(param.data)

    for target_param, param in zip(target_critic.parameters(), critic.
       parameters()):
        target_param.data.copy_(param.data)

    # Hadgalsan modeliig ashiglah

    if load_models:
        actor.load_state_dict(torch.load('./Models/' + str(0) + '_actor
           .pt'))
        critic.load_state_dict(torch.load('./Models/' + str(0) + '
           _critic.pt'))

        for target_param, param in zip(target_actor.parameters(), actor
           .parameters()):
            target_param.data.copy_(param.data)

        for target_param, param in zip(target_critic.parameters(),
           critic.parameters()):
            target_param.data.copy_(param.data)

        print("Models loaded!")
```

```
69
70      # Replay buffer uusgeh
71
72      ram = memory.ReplayBuffer(1000000)
73
74      #Buffer-g utgaar duurgeh (hot start)
75
76      st = np.float32(env.reset())
77      print(type(st))
78      for step in range(128):
79          action = env.action_space.sample()
80          new_observation, reward, done, info = env.step(action)
81
82          # Replay buffer-d state, action, reward, new_state -g hadgalah
83
84          ram.add(st, action, reward, new_observation)
85
86          if done:
87              st = env.reset()
88          else:
89              st = new_observation
90
91      print("Initial ram size: ", len(ram.buffer))
92
93      # Reward-iig hadgalah list
94
95      reward_list = []
96      average_reward_list = []
97      steps_reward_list = []
98
99      # Noise uusgeh
100
101     if NOISE_TYPE == "OU":
102         noise = utilities.OrnsteinUhlenbeckActionNoise(action_dimension
                )
103     elif NOISE_TYPE == "OUBaseline":
104         noise = utilities.OrnsteinUhlenbeckActionNoiseBaseline(mu=np.
                zeros(action_dimension), sigma=float(0.2))
105     elif NOISE_TYPE == "Parameter" or NOISE_TYPE == "ParameterWithOU":
106         #Parameter noise-d zoriulsan actor
107         actor_copy = models.Actor(state_dimension, action_dimension,
                action_max)
108
109         #Parameter noise
110         parameter_noise = utilities.AdaptiveParamNoiseSpec(
                initial_stddev=0.05,desired_action_stddev=0.3,
                adaptation_coefficient=1.05)
111         noise = utilities.OrnsteinUhlenbeckActionNoise(action_dimension
                )
112         #noise = utilities.OrnsteinUhlenbeckActionNoiseBaseline(mu=np.
                zeros(action_dimension), sigma=float(0.2))
113
```

```
    print("Noise type: ", NOISE_TYPE)

    start_time = time.time()

    tmp_noise_type = NOISE_TYPE

    for ep in range(MAX_EPISODE):

        # Anhnii state-g awah

        observation = env.reset()

        ep_reward = 0
        step_cntr = 0

        #Ehnii 20 episode uncorrelated noise-toigoor ywna
        if ep < 20:
            NOISE_TYPE = "NoNoise"
        else:
            NOISE_TYPE = tmp_noise_type

        if NOISE_TYPE == "Parameter":
            #Actor-g actor_copy-d huulah

            for target_param, param in zip(actor_copy.parameters(),
                actor.parameters()):
                target_param.data.copy_(param.data)

            # Parameter noise-iig neural suljeen deer nemeh

            parameters = actor_copy.state_dict()
            for name in parameters:
                parameter = parameters[name]
                rand_number = torch.randn(parameter.shape)
                parameter = parameter + rand_number * parameter_noise.
                    current_stddev

        for step in range(MAX_STEPS):
            env.render()
            state = np.float32(observation)

            # Action-g songoh

            tmp_state = Variable(torch.from_numpy(state))
            action_without_noise = actor.forward(tmp_state).detach()


            if NOISE_TYPE == "NoNoise":
                action = np.clip(action_without_noise.data.numpy(),
                    -1., 1.)
            elif NOISE_TYPE == "OU":
                #OU processiin noisetoi action
```

```
                action = np.clip(action_without_noise.data.numpy() + (
                    noise.sample() * action_max), -1., 1.)
            elif NOISE_TYPE == "OUBaseline":
                #OU Baseline noisetoi action
                action = np.clip(action_without_noise.data.numpy() +
                    noise(), -1., 1.)
            elif NOISE_TYPE == "Parameter":
                action = actor_copy.forward(tmp_state).detach().numpy()
            elif NOISE_TYPE == "ParameterWithOU":
                noise.reset()
                action_with_parameter_noise = actor_copy.forward(
                    tmp_state).detach()
                #Parameter noisetoi action
                action = action_with_parameter_noise.numpy() + (noise.
                    sample() * action_max)
                #action = np.clip(action_with_parameter_noise.numpy() +
                    noise(), -1., 1.)
            elif NOISE_TYPE == "Uncorrelated":
                #[-0.2, 0.2] random noisetoi action
                action = np.clip(action_without_noise.data.numpy() + (
                    np.random.uniform(-0.2,0.2) * action_max), -1., 1.)
            else:
                raise RuntimeError('Buruu turliin noise: "{}"'.format(
                    NOISE_TYPE))

            # Action-g hiij shine state, reward awah

            new_observation, reward, done, info = env.step(action)

            if done:
                new_state = None
            else:
                new_state = np.float32(new_observation)

                # Replay buffer-d state, action, reward, new_state -g
                    hadgalah

                ram.add(state, action, reward, new_state)
                ep_reward += reward
                steps_reward_list.append(reward)

            observation = new_observation

            # Replay buffer-aas 128 bagts turshalagiig random-oor awna

            states, actions, rewards, next_states = ram.sample_exp(128)

            states = Variable(torch.from_numpy(states))
            actions = Variable(torch.from_numpy(actions))
            rewards = Variable(torch.from_numpy(rewards))
            next_states = Variable(torch.from_numpy(next_states))

```

```
            # Critic network-g surgah

            predicted_action = target_actor.forward(next_states).detach
                ()
            next_val = torch.squeeze(target_critic.forward(next_states,
                 predicted_action).detach())
            y_expected = rewards + 0.99*next_val
            y_predicted = torch.squeeze(critic.forward(states, actions)
                )

            # Critic network-g shinechleh, critic loss-g tootsooloh

            critic_loss = F.smooth_l1_loss(y_predicted, y_expected)
            critic_optimizer.zero_grad()
            critic_loss.backward()
            critic_optimizer.step()

            # Actor network-g surgah

            predicted_action = actor.forward(states)
            actor_loss = -1*torch.sum(critic.forward(states,
                predicted_action))
            actor_optimizer.zero_grad()
            actor_loss.backward()
            actor_optimizer.step()

            # Target network-g shinechleh

            for target_param, param in zip(target_actor.parameters(),
                actor.parameters()):
                target_param.data.copy_(target_param.data * (1.0 -
                    0.001) + param.data * 0.001)

            for target_param, param in zip(target_critic.parameters(),
                critic.parameters()):
                target_param.data.copy_(target_param.data * (1.0 -
                    0.001) + param.data * 0.001)

            if done:
                break

            step_cntr += 1

        if NOISE_TYPE == "Parameter":
            #Noisetoi actor deer hiigdsen data-g list-d hadgalj awaad
                suuliin episode-d hiigdsen stepiin toogoor datagaa awna

            noise_data_list = list(ram.buffer)
            noise_data_list = np.array(noise_data_list[-step_cntr:])

            actor_copy_state, actor_copy_action, _, _ = zip(*
                noise_data_list)
```

```
249
250             #Noisetoi actoriin action
251             actor_copy_actions = np.array(actor_copy_action)
252
253             #Engiin actoriin action
254
255             actor_actions = []
256             for state in np.array(actor_copy_state):
257                 state = Variable(torch.from_numpy(state))
258                 action = actor.forward(state).detach().numpy()
259                 actor_actions.append(action)
260
261             #Distance tootsoh
262             diff_actions = actor_copy_actions - actor_actions
263             mean_diff_actions = np.mean(np.square(diff_actions),axis=0)
264             distance = math.sqrt(np.mean(mean_diff_actions))
265
266             #Sigma-g update hiih
267             parameter_noise.adapt(distance)
268
269         # reward-g hadgalj awna
270
271         reward_list.append(ep_reward)
272         average_reward = np.mean(reward_list[-40:])
273         print("Episode: {} Average Reward: {}" .format(ep,
                average_reward))
274         average_reward_list.append(average_reward)
275
276         #Model-iig hadgalah
277         if ep % 100 == 0 and ep != 0:
278             folder_path = './Models_ep1200_' + str(NOISE_TYPE) + '_' +
                    str(ACTOR_LEARNING_RATE) + '_' + str(
                    CRITIC_LEARNING_RATE)
279             if not os.path.exists(folder_path):
280                 os.makedirs(folder_path)
281
282             torch.save(target_actor.state_dict(), folder_path + '/' +
                    str(ep) + '_actor.pt')
283             torch.save(target_critic.state_dict(), folder_path + '/' +
                    str(ep) + '_critic.pt')
284
285             #Ram hadgalah
286             file_name = folder_path + '/' + str(ep) + '_ram.deque'
287             open_file = open(file_name, "wb")
288             pickle.dump(ram.buffer, open_file)
289             open_file.close()
290
291             #Average reward list hadgalah
292             file_name = folder_path + '/' + str(ep) + '_average_rewards
                    .list'
293             open_file = open(file_name, "wb")
294             pickle.dump(average_reward_list, open_file)
```

```
            open_file.close()

            #Step bolgonii rewardiig hadgalah
            file_name = folder_path + '/' + str(ep) + '_step_rewards.
               list'
            open_file = open(file_name, "wb")
            pickle.dump(steps_reward_list, open_file)
            open_file.close()

            print("Target actor, critic models saved")

        gc.collect()

    execution_time = time.time() - start_time
    file_name = './Models_ep1200_' + str(NOISE_TYPE) + '_' + str(
       ACTOR_LEARNING_RATE) + '_' + str(CRITIC_LEARNING_RATE) + '/' +
       str(ep) + '_execution_time.sec'
    open_file = open(file_name, "wb")
    pickle.dump(execution_time, open_file)
    open_file.close()
    # Reward-g durslen haruulah

    print("Reward max: ", max(average_reward_list))

    # plt.plot(average_reward_list, label = NOISE_TYPE)
    # plt.legend()
    # plt.xlabel("Episode")
    # plt.ylabel("Average Episode Reward")
    # plt.show()

    #os.system("shutdown /s /t 1")
```

Код А.4: main.py