**Neural network assignment 5- sarah hosseini 400222026**

-۱

در RL هدف ما این است که میانگین ریوارد تجمعی را ماکسیمایز کنیم. یک MDP اگر با پارامترهای زیر داشته باشیم:

S: حالت ها

A: اکشن ها

P(s'|s,a): احتمال انتقال (from s to s' given action a)

R(s,a): تابع توزیع ریوارد

γ: دیسکانت فکتور

میخواهیم یک $\pi_\theta(a|s)$ پیدا کنیم که cumulative discounted reward بیشینه شود یعنی

$$J(\theta) = E_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right]$$

طبق قضیه policy gradient ، گرادیان این عبارت نسبت به θ بصورت زیر است:

$$\nabla_\theta J(\theta) = E_{\pi_\theta}\left[\nabla_\theta log\pi_\theta(a|s) Q^{\pi_\theta}(s,a)\right]$$

که در آن $Q^{\pi_\theta}(s,a)$ همان تابع مقدار ریوارد است یعنی

$$\begin{aligned}
\nabla \mathbb{E}_\pi [r(\tau)] &= \nabla \int \pi(\tau) r(\tau) d\tau \\
&= \int \nabla \pi(\tau) r(\tau) d\tau \\
&= \int \pi(\tau) \nabla \log \pi(\tau) r(\tau) d\tau \\
\nabla \mathbb{E}_\pi [r(\tau)] &= \mathbb{E}_\pi [r(\tau) \nabla \log \pi(\tau)]
\end{aligned}$$

این قضیه کمک میکند که بصورت مستقیم بتوان پارامترهای policy را اپتیمایز کرد بدون اینکه از تابع value برای اپتیمایز کردن policy استفاده کنیم. برای مواقعی که دیتا ابعاد بالا دارد یا فضای اکشن ها پیوسته است مناسب است. در این حالات ما اصلا environment را مدل نمیکنیم پس این ها بسیار مناسب برای الگوریتمهای model free هستند.

-۲

در این روشها ما همزمان هم بدنبال پالیسی (اکتور) و هم ولیو (کریتیک) هستیم.

اکتور باید پالیسی بهینه را یاد بگیرد یعنی بهترین مپینگ از حالات به اکشنها.

کریتیک باید ولیو فانکشن را یاد بگیرد که تخمینی از میانگین مقادیر ریوارد آینده برای یک جفت حالت و اکشن  است.

این که از هر دوی این ها استفاده کنیم باعث بهبود میشود. مدلهای ولیو بیسد مثل کیو لرنینگ، میتوانند unstable باشند مخصوصا وقتی فضای حالات و اکشن ها پیوسته باشد و محیط پیچیده باشد. اینکه کریتیک ، اکتور را راهنمایی میکند و اکتور صرفا بر اساس ریواردهای نویزدار محیط جلو نمیرود، باعث بهبود مدل میشود.

روشهای پالیسی بیسد همچنین با تعداد اینترکشنهای کمتری، میتوانند پالیسی را بهینه کنند و روش اکتورکریتیک نیز این خوبی را دارد. همچنین باعث تریدآف خوبی بین اکسپلور و اکسپلویت میشود.

در واقع کریتیک به اکتور فیدبک میدهد و همچنین اکشن های اکتور برای اپدیت شدن به کریتیک داده میشوند.

-۳

هر دو الگوریتم از تابع advantage استفاده می کنند که میزان بهتر یا بدتر بودن یک اکشن را در مقایسه با اکشن میانگین در یک حالت مشخص می سنجد.

از لحاظ سنکرون بودن، a2c سنکرون است یعنی فقط یک اکتور دارد اما a3c آسنکرون است یعنی چندین اکتور همزمان و موازی کار میکنند و همین باعث اکسپلور بهتر میشود.

در A2C، شبکه های اکتور و کریتیک پس از هر تعامل با محیط به روز می شوند. در A3C، هر اکتور موازی با یک کپی از محیط تعامل دارد و گرادیان ها را در متغیرهای لوکال جمع می کند. این گرادیان ها به طور آسنکرون درون شبکه های اکتور و کریتیک گلوبال به روز می شوند که منجر به همگرایی سریعتر می شود.

همچنین A2C می تواند به دلیل ماهیت سنکرون آموزش از واریانس بالایی در به روز رسانی پالیسی خود رنج ببرد. این می تواند منجر به اکسپلور نابهینه محیط شود. از طرفی، A3C از تجربیات متنوع اکتورهای موازی سود می برد که منجر به تعادل بهتر اکسپلور و اکسپلویت می شود.

-۴

۱. ایجنت باید از یک مجموعه بی نهایت از اکشن ها انتخاب کند. این ابعاد بالا در فضا مسئله را بطور نمایی پیچیده میکند.

۲. تریداف اکسپلور و اکسپلویت در محیطهای پیوسته سخت تر است.

۳. تقریب زدن توابع ولیو و پالیسی در محیط پیوسته، پیچیدگی محاسباتی بالاتری دارد و مسئله ناپایداری و عدم همگرایی بیشتر دیده میشود و به هایپرپارامترها هم حساستر هستند.

۴. باید یاد بگیرد که بین اکشن هایی که داشته، اینترپولیت کند چون احتمالا یک اکشن را دوبار مواجه نخواهد شد. یافتن یک جنرالیزیشن خوب اینجا مهمتر و سختتر است.

راه حل ها میتواند از جمله زیر باشد:

۱.فضای اکشن را بصورت یک تابع پارامتری نشان دهیم مثلا یک توزیع گاوسی یا یک شبکه. ایجنت هم پارامترها را یاد میگیرد و بدین صورت مسئله راحتتر میشود.

۲. مسئله را تقسیم به زیرمسائلی بطور سلسله مراتبی کنیم که در سطح بالا، تصمیمات در فضای اکشن گسسته گرفته میشوند و در سطوح پایین به اکشن های پیوسته میرسیم.

۳.روشهای پالیسی گرادیان که گفتیم نیزبرای این فضاها مناسبند چون بطور مستقیم پالیسی را بهینه سازی میکنند.

۴. روشهای انسامبل نیز میتوانند توانایی ایجنت را بهبود دهند.

-۵

این الگوریتم یک نوع روشpolicy gradient برای فضاهای اکشن پیوسته است.

تراست ریجن اینجا مجموعه ای از همه‌ی آپدیت‌های شدنی (feasible) روی پالیسی در هر گام است. هدف، یافتن اپدیتی است که expected return را بیشینه کند اما در عین حال از پالیسی کنونی  بیش از حد واگرا نشود.

- Pseudocode:

**for** iteration=1, 2, . . . **do**

Run policy for $T$ timesteps or $N$ trajectories

Estimate advantage function at all timesteps

$$\underset{\theta}{\text{maximize}} \sum_{n=1}^{N} \frac{\pi_\theta(a_n \mid s_n)}{\pi_{\theta_{\text{old}}}(a_n \mid s_n)} \hat{A}_n$$

$$\text{subject to} \quad \overline{\text{KL}}_{\pi_{\theta_{\text{old}}}}(\pi_\theta) \leq \delta$$

**end for**

این کار باعث میشود اپدیت های پالیسی پایدارتر شود و تغییرات زیاد و نویزی نداشته باشد. (تغییرات زیاد میتوانند سبب اورفیت شوند) لرنینگ smooth تر میشود و واریانس اپدیتها کاهش می یابد. همچنین اثبات میشود که با این کار، پالیسی بطور یکنوا بهبود می‌یابد یعنی با هر اپدیت یا بهتر میشود یا به همان خوبی قبل عمل میکند. باعث میشود حساسیت مدل به هایپرپارامترهایی مثل لرنینگ ریت نیز کمتر شود.

-۶

انتروپی میزان عدم اطمینان یا رندومنس را در توزیع احتمالی نشان میدهد و به معنی افزایش اکسپلور و تنوع زیادتر در اکشن هاست.

بدون رگولاریزیشن انتروپی، ممکن است که ایجنت با سرعت زیادی به یک پالیسی نابهینه همگرا شود و در یک اپتیمم محلی گیر بیفتد و نتواند اکشن های مختلفی را اکسپلور کند.

رگولاریزیشن انتروپی ایجنت را به سمت اکسپلور کردن فضای حالت و اکشن سوق میدهد.

همچنین باعث میشود ایجنت یک پالیسی استوکستیک را یاد بگیرد یعنی ایجنت اکشن ها را بطور احتمالاتی بر اساس expected reward شان انتخاب کند نه اینکه بطور قطعی صرفا اکشن با بیشترین ریوارد را انتخاب کند. این باعث کشف پالیسی های بهینه میتواند بشود.
همچنین smooth شدن اپدیتهای پالیسی را هم به همراه دارد که یادگیری را stable میکند.
همچنین زیاده روی در این رگولاریزیشن هم میتواند باعث شود اکسپلور بیش از حد شود و پالیسی بهینه را پیدا نکند.
در ابتدای یادگیری، رگولاریزیشن انتروپی بالاتر مناسبتر است و بعدتر کمتر باید شود تا اجازه اکسپلویت به ایجت بدهیم.

-۷

الگوریتم ppo یک روش پالیسی بیسد است که اجازه نمیدهد اپدیت های خیلی بزرگی در پالیسی داشته باشیم تا بطور استیبل تر به راه حل بهینه برسیم.
الگوریتم trpo هم همین کار را میکرد اما معایبی داشت. مثلا پیچیدگی بالا چون از روش های درجه دو استفاده میکرد. همین نیاز به مشتقات درجه دوم و چندین گرادیان گرفتن باعث ادرهد محاسباتی بالا میشود. همچنین به هایپرپارامترها نیز بسیار حساس بود مثلا سایز تراست ریجن و باید این هایپرپارامترها را تیون کنیم.

اما ppo این معایب را ندارد. دو نوع ppo داریم، یکی با تابع هدف clipped و یکی با یک پنالتی روی kl divergnce.

در نوع اول، میخواهیم که نسبت احتمال پالیسی ها تغییرات کمی داشته باشند.

در نوع دوم، یک kl بین پالیسی قدیم و جدید داریم و با آن ایجنت را جریمه میکنیم.

نوع اول ppo پیاده سازی ساده ای دارد و نیازی به مشتقات درجه دو و فرایند بهینه سازی سختی شامل line search نیز نیست. از نظر محاسبات بهینه تر است. پالیسی بطور استیبل اپدیت میشود. هایپرپارامترهای کمتری هم دارد و هایپرپارامترهای خودش هم آسان تر تیون میشوند.

-۸

در این محیطها، چندین ایجنت با هم تعامل دارند. هر ایجنت نیز دنبال پالیسی ای است که ریوارد تجمعی خودش را بیشینه کند.

ایجنت ها میتوانند با هم همکاری، رقابت یا هر دوی این موارد را داشته باشند. هر ایججنت هم مجموعه اکشن خودش را دارد. Marl میتواند به صورت Markov Game که یک جنرالیزیشن از mdp است هم نوشته شود.

- Two-player zero-sum Markov Game (2P-MG)

$$
\begin{aligned}
&(S, A, O, T, R) \\
&\quad A \;:\; \text{Set of actions for the agent} \\
&\quad O \;:\; \text{Set of actions for the opponent} \\
&\quad T \;:\; S \times A \times O \to PD(S) \\
&\quad R \;:\; S \times A \times O \to \Re \\
&\text{Goal}_{agent} \;:\; \pi \text{ that maximizes } E\left\{\sum_{j=0}^{\infty} \gamma^j r_{t+j}\right\}
\end{aligned}
$$

در مسائل marl ، محیط داینامیک است  چون بقیه ایجنت ها ممکن است تاثیر روی محیط بگذارند.

پالیسی بهینه یک ایجنت هم ممکن است با گذر زمان عوض شود اما معمولا در سینگل ایجنت، اینگونه نیست.

در بسیاری مواقع عم ایجنت ها از اعمال بقیه ایجنت ها خبر ندارند و باعث میشود لرنینگ سخت تر شود. اگر هم ایجنت ها بخواهند ارتباط بگیرند، نیاز به روش و پروتکل ارتباط دارند که به پیچیدگی مسئله می افزاید. معین کردن اینکه هر ایجنت در رسیدن به ریوارد چقدر نقش داشته اینجا یک مسئله است. مخصوصا در مسائلی که ایجنت ها با همکاری هم به ریواردی دست می یابند.

در همکاری ایجنتها باید یاد بگیرند که همکاری کنند و باید روابط اکشن هایشان با عملکرد کلی تیم را هم بفهمند.

در مسائل رقابتی هم باید اکشن های بقیه را پیشبینی و خنثی سازی کنند.

وقتی ایجنت زیاد شود، ابعاد حالات و اکشنها بطور نمایی بالا میرود.

همچنین همگرایی وقتی ایجنت ها چند تا هستند و منفعتشان هم با هم در تعارض است، سختتر است.

۹-

اتنشن، به ما امکان میدهد که به توکن ها اجازه دهیم اصطلاحا با هم حرف بزنند یعنی با هم ارتباط بگیرند و روابطشان را بفهمیم و بفهمیم که کدام توکن ها به هم ربط دارند. مزایای اتنشن

میتوان به موازی سازی اشاره کرد که rnn آن را ندارد. در واقع در ترنسفورمر کل دیتا یک باره به مدل داده میشود.

آر ان ان ها و حتی lstmها همچنین بخاطر ونیش یا اکسپلود شدن گرادیان، نمیتوانند خوب روابط دورادور را دربیاورند. اما ترنسفرمر ها میتوانند درهر جایی از جمله که باشیم به هر جایی توجه زیاد بدهند.

از معایب ترنسفرمرها: پیچیدگی محاسباتی (سلف اتنشن از مرتبه quadratic است)، استفاده زیاد از مموری، نیاز به حجم زیادی از دیتا (چون خودشان پارامترهای زیادی دارند)، هایپرپارامترهای بیشتری که معرفی میشوند مثل تعداد لایه، head، سایز امبدینگ،...

ترنسفرمرها همچنین برا آنلاین لرنینگ (وقتی دیتا بطور متوالی در حال ورود است و مدل باید در لحظه پارامترهایش را اپدیت کند) مناسب نیستند چون به کل جمله نیاز دارند تا اتنشن را حساب کنند.

-۱۰

۱. ضرب نقطه ای

Formula: Given two vectors q (query) and k (key), the dot-product similarity is calculated as:

$$similarity(q,k)=q \cdot k$$

در فرم اسکیل شده که در پیپر ترنسفرمر استفاده شده، این مقدار را بر ریشه ی dimensionality تقسیم میکنیم.

چون از ضرب ماتریسی قابل محاسبه است، بهینه و قابل موازی سازی است. در تسک های nlp به وفور استفاده میشود. ترجمه ماشینی، خلاصه سازی متن، مدلهای زبانی.

۲. شباهت کسینوسی

similarity(q,k)=(q · k)¥ // q //  // k //

کسینوس زاویه بین دو بردار. در جاهایی که جهت نسبی بردارها از اندازه‌شان مهم تر است مثلا در محاسبه مشابهت معنایی توکن ها. در تسک های nlp مثل information retrieval کاربرد دارد.

۳. Additive (Bahdanau) Attention

$$similarity(q, k) = v^T tanh(W_q q + W_k k + b)$$

اینجا از یک شبکه feed forward برای یافتن شباهت استفاده میکنیم.

از ضرب داخلی عادی flexible تر است. روابط غیرخطی را میتواند بین توکن ها دربیاورد. در مدلهای seq2seq برای ترجمه و تولید متن کاربرد دارد.

۳. Bilinear Attention

$$similarity(q, k) = q^T W k$$

مشابه حالت قبل، چون یک متغیر وزن لرنبل دارد، روابط غیرخطی پیچیده را یاد میگیرد.

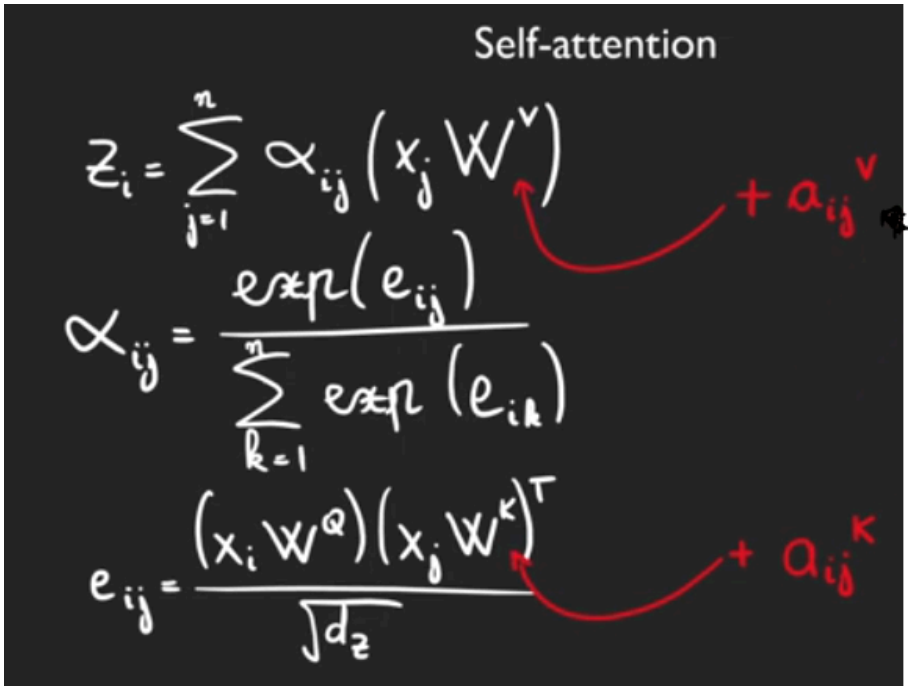
۵. Gaussian (RBF) Similarity

$$similarity(q, k) = exp(-(||q - k||^2)/(2\boldsymbol{\sigma}^2))$$

۶. در تسک های ویژن، برای یافتن شباهت بین پیکسلها در اتنشن، از کرنل‌های dilated استفاده میکنیم.

-۱۱

تکنیکی است برای انکد کردن پوزیشن توکن های داخل یک دنباله نسبت به همدیگر.
ترنسفرمرها ذاتا درکی از پوزیشن ندارند پس این کار ضروری است.  همچنین روشهای انکدینگ عادی (مثلا سینوسی) که به هر موقعیت یک Id خاص را میدهند، تاکیدی روی روابط توکن ها ندارند. پس انکدینگ ما اگر روابط را هم در نظر بگیرد، برای ما مفیدتر خواهد بود. همچنین چون روی روابط و فواصل نسبی تمرکز دارند، قدرت جنرالازیزیشن بیشتری روی دنباله های با طول متفاوت دارند و به شیفت خوردن پوزیشن کلمات هم جنرالایز میتوانند بکنند.
این نوع انکدینگ در سطح key و value انجام میشود.
اگر یک دنباله با سایز L داشته باشیم، یک ماتریس پوزیشن به نام R با سایز L*L میسازیم که Rij پوزیشن نسبی عنصر i ام به عنصر j ام را نشان میدهد.
سپس مقادیر ماتریس هر یک به یک وکتور امبد میشوند که میتواند توسط یک لایه امبدینگ لرنبل و با همان مکانیزم اتنشن انجام گیرد.




-۱۲

داشتن چندین هد مختلف، به مدل اجازه میدهد که کانتکست های مختلف را از توکن ها ید بگیرد. قطعا یک هد، حداکثر میتواند یک کانتکست را دربیاورد ولی با مولتی هد چندین فیچر را میتوان درآورد. این گونه مدل میتواند یک representation بهتر از دیتا را یاد بگیرد. مثلا با یک هد، از لحاظ دستوری روابط را بفهمد، با هد دیگر از لحاظ جنسیت کلمات (مثلا رابطه تضاد در ملکه و شاه)، با یک هد دیگر از لحاظ سلطنتی بودن(مثلا رابطه تناسب در ملکه و شاه) و …

همچنین قابلیت موازی سازی بالایی به ما میدهد چون هر هد موازی هدهای دیگر کار میکند و پردازش سریع و بهینه تر میشود.

از مسکد مولتی هد وقتی استفاده میکنیم که نخواهیم مدل همه ورودی را ببیند. مثلا موقع پیشبینی کلمه بعدی در یک جمله، مدل اصلا نباید توکنهای بعدی را ببیند تا دیتا لیک نشود.

همچنین باعث میشود که ارتباط علی و معلولی در جملات حفظ شود که هر کلمه فقط به کلمات قبلی خودش وابسته است نه بعدی.

مسک کردن با ضرب یک ماتریس که در قسمت بالای قطر اصلی آن $-\infty$ گذاشته ایم، بدست می‌آید. (چون بعد از سافتمکس، مقدار آنها صفر خواهد شد)

در ترنسفرمرها، از لیر نورم برای پایدار شدن و سریع شدن یادگیری، استفاده میشود. از آن روی ورودی لایه‌های شبکه استفاده میکنیم. با این کار از ونیش/اکسپلود گرادیان جلوگیری میشود.

باعث میشود واریانس عوض نشود و همگرایی سریعتر شود. قدرت جنرالیزیشن مدل نیز بالا میرود چون به واریانس ورودی مقاوم میشود.

برخلاف Batch Normalization که ورودی ها را در راستای بچ نورمال می کند، Layer Normalization نرمال سازی را در راستای فیچرها در یک لایه انجام می دهد.

الگوریتم لیرنورم:

۱-میانگین ($\mu$) و انحراف استاندارد ($\sigma$) خروجی های لایه را در راستای همه‌ی فیچرها محاسبه کن.

۲-میانگین را از هر فیچر کم کن: $x_i = x_i - \mu$

۳-هر فیچر را بر انحراف استاندارد تقسیم کن: $x_i = x_i / \sigma$

۴-یک ضریب مقیاس (γ) و یک ضریب شیفت (β) را برای داشتن مقداری تغییرپذیری اعمال کن: $x\_i = \gamma * x\_i + \beta$

برای هر سمپل ترین، جداگانه اعمال میشود. برای ورودی x با بعد d داریم:

$$LayerNorm(x)=\gamma((x-\mu)/\sigma)+\beta$$

در ترنسفرمرها اگر قبل/بعد از مولتید اتنشن یا فیدفوروارد یا لایه دیگری، لیرنورم بگذاریم، به آن pre-norm و post-norm گفته میشود.