Policy Gradient utilizando CUDA aplicado al juego Pong de ATARI

Amaury Gutiérrez Acosta, Ixchel G. Meza Chávez

May 2, 2017

Objetivo

Nuestro objetivo es crear un agente que pueda aprender a jugar el videojuego Pong de Atari a partir de la secuencia de matrices de pixeles que constituyen el juego. Para lograr esto se propone el uso de una red neuronal. La entrada de la red sería la matriz de pixeles y la salida es un elemento del espacio de acciones que el agente puede tomar. Para realizar el aprendizaje por refuerzo para nuestro problema de interés usamos el algoritmo policy gradient, el cual está inspirado en uno de los métodos clásicos de optimización, que es descenso en gradiente estocástico.

Introducción

Hay dos tipos de problemas de optimización que surgen en machine learning:

- problemas de optimización convexa -derivados de el uso de regresión logística o support vector machines
- problemas no convexos y altamente no lineales -derivados del uso de redes neuronales DNN

En el contexto de machine learning a gran escala se ha tenido gran interés la propuesta de Robbins y Monro de algoritmos estocásticos, como el método de gradiente estocástico SG.

En DNN se describe una función de predicción h cuyo valor es calculado aplicando transformaciones sucesivas a un vector de entrada x. Estas transformaciones son capas o layers. Por ejemplo una capa canónica conectada completamente realiza un cálculo donde x es el vector de entrada, la matriz W y el vector b contienen los parámetros de la capa y s es una función de activación no lineal component-wise.

$$x^{(j)} = s(W_j x^{(j-1)} + b_j)$$

La función sigmoidea $s(x) = \frac{1}{(1+exp(-x))}$ y la función hinge $s(x) = max\{0, x\}$ conocida como ReLU son funciones populares para utilizarse como función de activación s.

El problema de optimización en este contexto involucra un training set $(x_1, y_1)...(x_n, y_n)$ y la elección de una función de pérdida l de tal forma que

$$min \frac{1}{n} \sum_{i=1}^{n} l(h(x_i; w), y_i)$$

Como ya se mencionó dicho problema es no convexo altamente no lineal, sin embargo se han calculado soluciones aproximadas por medio de métodos basados en el gradiente, pues el gradiente del objetivo en la ecuación anterior con respecto al vector parámetro w se puede calcular por medio de la regla de la cadena usando diferenciación algorítmica. Ésta técnica de diferenciación es conocida como *back propagation*.

Métodos de optimización

Los problemas de optimización en machine learning surgen a través de la definición de las funciones de predicción y de pérdida que aparecen en medidas de riesgo esperado y empírico que intentamos minimizar.

Asumimos que la función de predicción h es de forma fija y está parametrizada por el vector real $w \in \mathbb{R}^d$ sobre el cual se realiza la optimización. Formalmente para una funcion dada $h(.,.) : \mathbb{R}^{d_x} \times \mathbb{R}^d \to \mathbb{R}^{d_y}$ consideramos la familia de funciones de predicción

$$H := \{h(.; w) : w \in \mathbb{R}^d\}$$

de tal forma que intentamos encontrar la función de predicción en esta familia que minimiza las pérdidas debido a predicciones inexactas. Para esto asumimos una función de pérdida $l : \mathbb{R}^{d_y} \times \mathbb{R}^{d_y} \to \mathbb{R}$ donde dado el par de entrada-salida (x, y) produce la pérdida $l(h(x; w), y)$ cuando $h(x; w)$ y y son salidas predecidas y verdaderas respectivamente.

En un escenario ideal, el vector parámetro w se escoge para minimizar la pérdida esperada para cualquier par entrada-salida. Es decir, que asumimos que las pérdidas son medidas con respecto a una distribución de probabilidad P(x,y) lo cual representa la verdadera relación entre entrada y salida. Es decir que asumimos que el espacio entrada-salida $\mathbb{R}^{d_x} \times \mathbb{R}^{d_y}$ es creado con $P : \mathbb{R}^{d_x} \times \mathbb{R}^{d_y} \to [0, 1]$ y la función objetivo que queremos minimizar es

$$R(w) = \int_{\mathbb{R}^{d_x} \times \mathbb{R}^{d_y}} l(h(x; w), y) dP(x, y) = \mathbb{E}[l(h(x; w), y)]$$

Decimos que $R : \mathbb{R}^d \to \mathbb{R}$ produce el riesgo esperado (pérdida esperada) dado un vector parámetro w con respecto a la distribución de probabilidad P.

Si no se conoce P no se puede intentar minimizar la ecuación anterior, por lo que en la práctica se busca la solución a un problema que involucra un estimado del riesgo esperado R.

Como en aprendizaje supervisado se tiene acceso a un set de muestras entrada-salida de $n \in \mathbb{N}$, el cual es $\{(x_i, y_i)\}_{i=1}^n \subseteq \mathbb{R}^{d_x} \times \mathbb{R}^{d_y}$ con el cual se puede definir la función de riesgo empírico $R_n : \mathbb{R}^d \to \mathbb{R}$ por medio de

$$R_n(w) = \frac{1}{n} \sum_{i=1}^{n} l(h(x_i; w), y_i)$$

Por lo general la minimización de R_n es considerado como el problema práctico de optimización de interés.

Método batch de gradiente

El método más simple es el algoritmo *steepest descent* o *gradiente batch* definido con la iteración

$$w_{k+1} \leftarrow w_k - \alpha_k \nabla R_n(w_k) = w_k - \frac{\alpha_k}{n} \sum_{i=1}^{n} \nabla f_i(w_k)$$

donde el cálculo del paso $-\alpha_k \nabla R_n(w_k)$ es más costoso que el del método estocástico $-\alpha_k \nabla f_{i_k}(w_k)$, sin embargo se podría esperar un mejor paso cuando se consideran todas las muestras en la iteración.

Al utilizar métodos tipo batch se tienen a disposición una serie de técnicas de optimización no lineal que han sido desarrolladas a lo largo de varias décadas, incluyendo los métodos full gradient, gradiente acelerado, gradiente conjugado, quasi-Newton y Newton inexacto. Además gracias a la estructura de suma de R_n el método batch puede ser paralelizado pues la mayoría de los cálculos son para evaluar R_n y ∇R_n.

Método estocástico de gradiente

En el contexto de minimizar el riesgo empírico R_n con $w_1 \in \mathbb{R}^d$ el método estocástico de gradiente SG se define como

$$w_{k+1} \leftarrow w_k - \alpha_k \nabla f_{ik}(w_k)$$

donde para todo $k \in \mathbb{N} := \{1, 2, ...\}$, el índice i_k, correspondiente a la seed $\xi[i_k]$ como el par muestra (x_{i_k}, y_{i_k}), es escogido aleatoriamente de $\{1, ..., n\}$ y α_k es un paso positivo. Cada iteración de este método involucra sólo el cálculo del gradiente $\nabla f_{ik}(w_{ik})$ correspondiente a una muestra. A diferencia de un algoritmo de

optimización determinística, en éste método la secuencia de iteración no está determinada únicamente por la función R_n, el punto de inicio w_1 y el paso de la secuencia $\{\alpha_k\}$, sino que $\{\alpha_k\}$ es un proceso estocástico cuyo comportamiento está determinado por la secuencia aleatoria $\{i_k\}$. Aunque cada dirección $-\nabla f_{i_k}(w_k)$ pueda no ser descendiente de w_k al producir una derivada direccional negativa para R_n de w_k, si es una dirección descendiente *en espera*, entonces la secuencia $\{w_k\}$ puede ser guiada hacia un mínimo de R_n.

En general la función objetivo es el riesgo esperado o empírico

$$F : \mathbb{R}^d \to \mathbb{R}, R(w) = \mathbb{E}[f(w;\xi)] or R_n(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w)$$

Tanto para el método de gradiente estocástico como para el tipo batch aplica el análisis, la única diferencia es la forma en que se escogen los estimados del gradiente estocástico en el método.

El algoritmo es el siguiente:

1. Escoger una iteración inicialw_1
2. Para $k = 1, 2, ...$ hacer
    3. Generar una realización de una variable aleatoria ξ_k
    4. Calcular un vector estocástico $g(w_k, \xi_k)$
    5. Escoger un tamaño de paso $\alpha_k > 0$
    6. Establecer la nueva iteración como $w_{k+1} \leftarrow w_k - \alpha_k g(w_k, \xi_k)$
3. fin de bucle

Se deben considerar varios aspectos del algoritmo general: primero, el valor de la variable aleatoria ξ_k se considera sólo como una semilla para generar una dirección estocástica, por lo que escogerla puede verse como la elección de una muestra de entrenamiento como en el método simple de SG o puede representar un set de muestras como en el método mini-batch SG. Segundo, $g(w_k, \xi_k)$ puede representar un gradiente estocástico como por ejemplo un estimador imparcial de $\nabla F(w_k)$ como en el método clásico de Robbins y Monro, o puede representar una dirección estocástica Newton o quasi-Newton. Por lo que $g(w_k, \xi_k)$ puede ser

$$\nabla f(w_k; \xi_k)$$

o

$$\frac{1}{n_k}\sum_{i=1}^{n_k} \nabla f(w_k; \xi_{k,i})$$

o

$$H_k \frac{1}{n_k}\sum_{i=1}^{n_k} \nabla f(w_k; \xi_{k,i})$$

Tercero, el algoritmo de SG permite varias opciones de la secuencia de tamaños de pasos $\{\alpha_k\}$, puede ser fija o reducir paulatinamente. Por último, el algoritmo de SG cubre técnicas de aprendizaje activo donde la iteración w_k influye en la selección de la muestra.

Complejidad computacional

Cuando se va a ejecutar un algoritmo para aprendizaje para grandes cantidades de datos la complejidad computacional juega un papel importante y es un factor limitante. Se necesitan algoritmos de aprendizaje que escalen de forma aproximadamente lineal con el volumen total de datos así como con su tiempo de cómputo.

Los mejores algoritmos de optimización no son necesariamente los mejores algoritmos de aprendizaje. Incluso se puede decir que ciertos algoritmos se desempeñan bien independientemente de la razón asumida para el error estadístico de estimación.

Backpropagation

Es un algoritmo muy popular en el aprendizaje de redes neuronales, porque es conceptualmente simple, eficiente computacionalmente y porque si funciona normalmente.

Al diseñar y entrenar una red que usa backpropagation se debe escoger arbitrariamente el número de nodos, capas, razones de entrenamiento, sets de entrenamiento y de prueba entre otras. Estas elecciones son críticas y dependen en gran medida del problema y de los datos disponibles.

La forma más simple de machine learning multicapas con aprendizaje basado en gradiente es un conjunto de módulos, cada uno de los cuales implementa una función $X_n = F_n(W_n, X_{n-1})$ donde X_n es un vector representando la salida del módulo, W_n es el vector de parámetros modificables en el módulo y X_{n-1} es el vector de entrada al módulo. La entrada X_0 al primer módulo es el patrón de entrada Z^P. Si la derivada parcial de E^P con respecto de X_n es conocida, entonces la derivada parcial de E^P con respecto de W_n y X_{n-1} se puede calcular usando la recurrencia backward

$$\frac{\partial E^P}{\partial W_n} = \frac{\partial F}{\partial W}(W_n, X_{n-1})\frac{\partial E^P}{\partial X_n}$$

y

$$\frac{\partial E^P}{\partial X_{n-1}} = \frac{\partial F}{\partial X}(W_n, X_{n-1})\frac{\partial E^P}{\partial X_n}$$

donde $\frac{\partial F}{\partial W}(W_n, X_{n-1})$ es el jacobiano de F respecto de W evaluado en el punto (W_n, X_{n-1}) y $\frac{\partial F}{\partial X}(W_n, X_{n-1})$ es el jacobiano de F respecto de X. Cuando dichas ecuaciones se aplican a los módulos en reversa, es decir desde la capa N a la 1, se pueden calcular todas las derivadas parciales de la función de pérdida con respecto de todos los parámetros. La forma de calcular los gradientes se conoce como backpropagation.

El caso de redes neuronales multicapas es un caso especial del sistema anterior, donde los módulos son capas alternadas de multiplicaciones de matrices (los pesos) y función sigmoidea component-wise (las unidades), es decir

$$Y_n = W_n X_{n-1}$$
$$X_n = F(Y_n)$$

donde W_n es una matriz cuyo número de columnas es la dimensión de X_{n-1} y el número de renglones es la dimensión de X_n. F es una función vector que aplica la función sigmoidea a cada componente de su entrada. Y_n es el vector de sumas pesadas o entradas totales a la capa n. Aplicando la regla de la cadena a la ecuación anterior, las ecuaciones clásicas de backpropagation en su forma matricial son:

$$\frac{\partial E^P}{\partial Y_n} = F'(Y_n)\frac{\partial E^P}{\partial X_n}$$
$$\frac{\partial E^P}{\partial W_n} = X_{n-1}\frac{\partial E^P}{\partial Y_n}$$
$$\frac{\partial E^P}{\partial X_{n-1}} = W_n^T\frac{\partial E^P}{\partial Y_n}$$

La forma más simple de aprendizaje (minimización) es el algoritmo de descenso de gradiente, donde W se ajusta iterativamente de la siguiente manera:

$$W(t) = W(t-1) - \eta\frac{\partial E}{\partial W}$$

En el caso más sencillo η es una constante escalar. En caso más sofisticados η es variable. En otros métodos *eta* es de la forma de una matriz diagonal o es un estimado de la matriz Hessiana inversa de la función de pérdida, como en los métodos de Newton y Quasi-Newton.

En cada iteración de la ecuación anterior se requiere utilizar todo el dataset para calcular el promedio del gradiente verdadero, lo cual se hace utilizando el método batch, a diferencia del método estocástico, donde se

escoge un solo ejemplo $\{Z^t, D^t\}$ aleatoriamente del set de entrenamiento en cada iteración t, y se calcula un estimado del gradiente verdadero basado en el error E^t de ese ejemplo y entonces se actualizan los pesos de la siguiente forma

$$W(t+1) = W(t) - \eta \frac{\partial E^t}{\partial W}$$

Debido a que la estimación del gradiente es ruidosa, los pesos podrían no moverse precisamente por el gradiente en cada iteración, lo cual puede tener ventajas, pues es común que redes no lineales tengan varios mínimos locales de diferentes profundidades, por lo que con el ruido los pesos podrían saltar de un mínimo a otro y encontrar uno más profundo. Las ventajas de aprendizaje estocástico son que es mucho más rápido que el parendizaje tipo batch, muchas veces tiene mejores resultados y se puede usar para rastrear cambios.

Policy gradient

El método policy gradient trata de aprender una función policy. Se tiene que aprender a partir de trayectorias generadas a partir de la policy actual. Una forma de aprender una policy de aproximación es maximizando la recompensa esperada usando métodos basados en el gradiente, por lo que en policy gradient el gradiente de la recompensa esperada respecto a los parámetros de la policy se puede obtener por medio del teorema de policy gradient para policies estocásticas.

Se pone de entrada el estado y de salida obtenemos una acción, luego se ve en que acciones se desemepñó bien y se incrementa su probabilidad.

Este tipo de métodos encuentran la policy óptima optimizando a largo plazo la recompensa. El método qe se usa para optimizar la recompensa es el SG.

Si intentamos aprender la policy, representamos la policy como la probabilidad de realizar una determinada acción dado el estado $\pi = P(a|s)$. Para espacios de acción de dimension alta se puede optar por una probabilidad aleatoria para escoger una acción, lo cual puede ser esencial para una estrategia óptima. Sin embargo el proceso de aprender una policy tiene varianza muy grande por lo que puede tomar mucho tiempo el entrenamiento y se puede llegar a un óptimo local. Para aprender una policy asumimos la función $\pi[w](s, a)$ que tiene como entradas la tupla estado-acción y da como salida una probabilidad. W representa parámetros que usamos para aprender esta función y son los pesos de la red neuronal. Si conocemos el gradiente de los parámetros para hacer una probabilidad de tomar una acción A del estado S más probable, ajustamos los parámetros en esa dirección a la escala de lo que se obtiene al realizar esa acción. Usando aprendizaje Monte-Carlo, para aprender una policy tenemos que esperar el final del episodio. Para cada paso se actualiza el parámetro W:

$$W = W + learningRate * [derivativeOfWToMaximizeP(a|s) * [totalReward]]$$

Notese que la recompensa total siginifca desde ese paso en adelante. Se tiene que tomar en cuenta que dado que esperamos hasta el final del episodio, la varianza es alta. Para resolver esto se puede hacer una evaluación de la policy para estimar la función value. Esto se conoce como actor-critic, donde el actor es la policy y la critic es la función value. Los pasos para realizara son los siguientes:

- Estimar la función value V(s) usando la evaluación de la policy de π
- Realizar una acción basada en la policy π
- Encontrar el error TD de esa acción
- actualizar los parametros W a la escala del error TD

$$TDerror = reward + discountFactor * V(nextState) - v(currentState)$$

Policy gradients es un caso especial de un estimador del gradiente de la función score. El caso general es que cuando tenemos la expresión $E_{x\sim p(x|\theta)}[f(x)]$ que puede ser la espectativa de una función score escalar $f(x)$

bajo una distribución de probabilidad $p(x;\theta)$ parametrizada por θ. $f(x)$ es la función de recompensa y $p(x)$ sería la red de policy, que es un modelo para $p(a|s)$, dada una distribución sobre acciones para cualquier estado o en nuestro caso imagen. Entonces estamos interesados en encontrar como desplazaríamos la distribución a través de los parámetros θ para incrementar los scores de las muestras, es decir como cambiamos los parámetros de la red de tal forma que las muestras de acciones obtengan recompensas más altas. Para esto tenemos la definición de la espectativa o esperanza:

$$\nabla_\theta E_x[f(x)] = \nabla_\theta \sum_x p(x)f(x) = \sum_x p(x)\nabla_\theta logp(x)f(x) = E_x[f(x)\nabla_\theta logp(x)]$$

Es decir que, nosotros tenemos una distribución $p(x;\theta)$ de la que podemos sacar muestras. Para cada muestra podemos evaluar la función score f, la cual toma la muestra y nos da un score escalar. Esta ecuación nos dice como tenemos que desplazar la distribución a través de sus parámetros θ si queremos que sus muestras logren scores altos de acuerdo a f. Entonces, se toman unas muestras x, se evaluan sus scores $f(x)$ y para cada x también se evalua el segundo término $\nabla_\theta logp(x;\theta)$, donde el segundo término es un vector -el gradiente que nos da la dirección en el espacio de parámetros que nos lleva a incrementar la probabilidad asignada a x. Es decir, si empujamos los parámetros en la dirección $\nabla_\theta logp(x;\theta)$ veremos que aumenta ligeramente la nueva probabilidad asignada a x. La fórmula nos dice que debemos tomar esa dirección y multiplicarla por el score escalar $f(x)$, esto producirá que las muestras que tienen un score alto jalarán más fuerte la densidad de probabilidad que las muestras de score bajo, de tal forma que si se hace una actualización basada en muchas muestras de p, la densidad de probabilidad se desplazará en la dirección de los scores altos, haciendo más probable que tengamos muestras con socre alto. Esto quiere decir que la forma de cambiar los parámetros de la policy es tomando el gradiente de las acciones muestreadas, multiplicarlo por el score y sumar todo.

Arcade-Learning-Environment: Plataforma de evaluación para agentes generales

La *Arcade Learning Environment (ALE)* es una plataforma para investigación de inteligencia artificial. Está basada en Stella, que es un emulador de Atari 2600 VCS.

Ale es un framework orientado a objetos con soporte para añadir agentes y juegos que permite el entrenamiento de agentes de aprendizaje por refuerzo, además su código es multiplataformas y los agentes programados en c++ tienen acceso a todas las características de ALE.

Cabe señalar que ALE permite la extracción automática del marcador y la señal de finalización de más de 50 juegos.

CUDA

Copute Unified Device Architecture (CUDA) es una plataforma de cómputo en paralelo de nVidia, que incluye un compilador y herramientas de desarrollo que permite programar en C, C++ y Fortran para usar los recursos del GPU.

Entre las ventajas de CUDA se encuentran que el código puede leer direcciones arbitrarias en la memoria lo que permite lecturas dispersas, además se cuenta con una memoria virtual unificada, y una memoria compartida por regiones que es compartida por los threads de un bloque, sin embargo la velocidad de la lectura entre la memoria del host y del device depende del ancho de banda del bus y hay un número máximo de threads por bloque, lo que puede repercutir en el diseño del código para realizar operaciones de grandes cantidades de datos.

Caso de estudio

Proponemos crear un agente que aprenda a jugar el videojuego Pong de Atari. Para evaluar dicho agente usamos la plataforma *Arcade Learning Environment (ALE)* la cual nos da como entradas imagenes de 210x160 pixeles en escala de grises y tenemos que decidir si el agente se mueve hacia arriba o hacia abajo. Una vez

que se decide la dirección del movimiento el simulador nos da una recompensa por cada decisión, la cual puede ser +1 si la pelota pasó al oponente, -1 si no le dimos a la pelota y 0 en cualquier otro caso.

Para esto usamos una red neuronal de dos capas que toma como entrada la imagen en pixeles de dimensiones 210x160 y da como salida la probabilidad de moverse hacia arriba. Para esto usamos policy estocástica. Cada iteración que será muestreada de esta distribución para obtener el movimiento realizado por el agente.

La policy forward se compone de los siguientes pasos una vez que se preprocessó el frame x:

- Calcular la activación de la capa escondida: $h = W_1 x$
- Calcular la capa ReLU que da no linearidad: threshold a 0 para h
- Calcular el logaritmo de la probabilidad de moverse hacia arriba: $logp = W_2 h$
- Calcular la funcion sigmoidea que da la probabilidad de ir hacia arriba $p = \frac{1}{1+e^{-logp}}$

donde W_1 y W_2 son dos matrices inicializadas aleatoriamente. La función sigmoidea aplasta la probabilidad de salida al rango [0,1]. Las neuronas de la capa escondida que tiene sus pesos en W_1 detectan varios escenarios del juego y los pesos en W_2 deciden para cada uno de esos escenarios hacia donde moverse.

En el preprocesamiento se obtiene la diferencia de dos frames consecutivos para de esa forma obtener el movimiento del juego.

Existe un problema llamado *credit assignment problem* en el cual no sabemos cual de todas las acciones del agente contribuyeron a la obtención de una recompensa positiva.

Policy Gradient: Una vez que la red policy calculó las probabilidades de moverse hacia arriba y hacia abajo, se hace una muestra de una acción de esa distribución, es decir se toma una decisión y se ejecuta en el juego. Podriamos inmediatamente por ejemplo poner como gradiante 1 para la acción escogida y encontrar el vector gradiente que insita a la red a realizar la acción elegida en el futuro. De esta forma se puede calcular el gradiente inmediatamente, sin embargo no sabemos si esa acción es una buena elección, por lo que esperamos hasta el final del juego y ver que tenemos de recompensa y una vez que tengamos 1 o -1 (ganar o perder el juego) ponemos ese número como gradiente a la acción que se realizó. ASí si ponemos la recompensa como logp del movimiento que se hizo y hacemos backpropagation encontraremos un gradiente que insita o no a la red a realizar esa acción para esa entrada en el futuro. De esta forma tenemos una policy estocástica, donde se muestrean las acciones y las acciones que llevan a una recompensa positiva son incentivadas en el futuro y al revés. Cuando se termina el juego se puede terminar con más de +1 o menos de -1.

Protocolo de entrenamiento: se inicializa la red policy con W_1 y W_2 aleatorias y se juegan 10 episodios de Pong. Vamos a asumir que cada juego contiene 200 frames, por lo que se toman 20,000 decisiones de moverse hacia arriba o hacia abajo y por cada una de estas conocemos el gradiente del parámetro, que nos dice cómo debemos cambiar los parámetros si queremos incentivar esa decisión en ese estado en el futuro. Por lo que tenemos que etiquetar cada decisión que se ha tomado como buena o mala, por lo que para las 200 decisiones de todos los juegos que se ganaron se le pone +1 al respectivo gradiente, se hace backpropagation y se actualiza el parámetro incentivando las acciones que escogimos en todo esos estados y se hace lo contrario es decir se pone -1 para las 200 acciones de los juegos que se perdieron.

Otra forma de ver este problema es especificando una función de pérdida como en aprendizaje supervisado, donde el objetivo es maximizar $\sum_i logp(y_i|x_i)$ donde x_i, y_i son ejemplos de entrenamiento como imágenes y sus etiquetas. Policy gradients es lo mismo excepto por dos diferencias:

- No tenemos las etiquetas correctas y_i, por lo que substituimos la acción que muestreamos de la policy cuando se ve x_i como "fake label"

- Modulamos la pérdida para cada ejemplo multiplicando por un valor basandonos en el resultado final dado que intentamos incrementar el logaritmo de la probabilidad log p de las acciones que siriveron e intentamos disminuirlo para las que no sirvieron.

En resumen nuestra función de pérdida es $\sum_i A_i logp(y_i|x_i)$ donde y_i es la acción que muestreamos y A_i es un número que llamamos como ventaja, que puede ser +1 si ganamos y -1 si perdemos el episodio. Esto asegura que maximizamos el logaritmo de la probabilidad de las acciones que nos llevan a ganar y minimiza

las que nos llevan a perder. Se podría decir que reinforcement learning es como aprendizaje supervisado pero en un dataset que cambia constantemente, escalado por la ventaja y queremos hacer pocas actualizaciones basados en cada dataset muestreado.

Al recibir en cada paso o frame una recompensa r_t es común utilizar una "recompensa rebajada" con lo cual la recompensa final sería $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ donde γ está en el rango entre 0 y 1 y se le llama el factor de rebaja. Nosotros escogimos como $\gamma = 0.99$. Esta expresión establece que la fuerza con la que incentivamos una acción muestreada es la suma ponderada de todas las recompensas posteriores, sin embargo estas recompensas son exponencialmente menos importantes. En la práctica tambien es importante normalizarlas. Una vez que se calculó R_t para todas las recompensas de los 10 juegos del batch, se estandariza, es decir, se resta el promedio y se divide por la desviación estándar antes de usarla en la backpropagation. De esta forma incentivamos o no aproximadamente la mitad de las acciones ejecutadas. Esta medida es una forma de controlar la varianza del estimador del gradiente policy.

Aprendizaje: Se entrenó una red policy de dos capas con 200 neuronas de capas escondidas usando RMSProp que es un optimizador que utiliza la magnitud del gradiente reciente para normailzar los gradientes en batches de 10 episodios, cada uno de los cuales tiene varios juegos, pues el juego acaba cuando algun jugador alcanza un score igual a 21.

Algoritmo

1. Se definen $H = 200$ como el número de neuronas de la red, $D = 80 * 80$ que es la dimensión de las imagenes de entrada, $ROWS = 80$, $COLS = 80$ $GAMMA = 0.99$ como el factor de rebaja, $BATCHSIZE = 10$ que es el número de episodios, los cuales constan de 21 juegos, $DECAYRATE = 0.99$ y $LEARNINGRATE = 0.001$
2. Se establecen los parámetros deseados para el juego Pong, que es una semilla aleatoria y 123
3. Se pone como false el display de la pantalla y del sonido
4. Se carga el archivo ROM y resetea el sistema para que los nuevos settings tengan efecto
5. Se obtiene el vector de acciones legales, las cuales pueden ser moverse hacia arriba o hacia abajo
6. Se obtienen altura y ancho de la pantalla del juego que son 210 y 160 respectivamente
7. se generan valores aleatorios para las matrices w1 (200x80*80) y w2 (200x1)
8. Se hace un loop infinito donde se realiza lo siguiente:
    1. se obtiene la pantalla en escala de grises y se almacena en *data*
    2. se hace le preprocesamiento de *data*, en donde se obtiene como datos interesantes de data un batch de 160*160. Se hace un recorrido por esos datos de interés. De los valores que se recorren se toman los multiplos de dos y se almacena un cero si el valor de los datos de interés es 87 el cual corresponde al color del fondo del juego o un uno si es otro valor que serían las barritas y la pelota del juego. Todo esto se guarda en la variable current.
    3. se resta current - last y se asigna a x
    4. se asigna a last, current
    5. se hace policy forward para obtener la probabilidad:
        - se multiplica h=w1*x (200x1)
        - se aplica la función de activación ReLU a h
        - se agrega h al final del vector hiddenstates
        - se multiplica logp=w2*h (1x1)
        - Se obtiene la probabilidad de que se mueva hacia arriba por medio de la función sigmoidea $prob = \frac{1.0}{1.0+e^{(-logp)}}$
    6. Se ejecuta getaction para escoger aleatoriamente la accion: moverse hacia arriba o hacia abajo
        - si número aleatorio/(valor máximo de ese número aleatorio + 1) < 0.5 regresa act=2, que de acuerdo a las variables de ALE significa "up""

- si no regresa act=3, que de acuerdo a ALE significa “right”, pero para el juego de pong es igual a “down”

7. Si act es 2 se incrementa up y fakelabel=1, si no se incrementa down y fakelabel=0
8. Se agrega x al final de exs
9. Se agrega al final de dlogps fakelabel-prob
10. se obtiene el objeto con la acción valida de ALE, Action a = legal_actions[act]
11. Se asigna a skip = (rand() % 3) + 2, es decir que es un numero en el rango de 0 a 2 + 2, es decir en el rango (2-4)
12. Se asigna 0 a reward
13. para un loop que puede correr entre 2 y 4 veces se suma a reward lo que salga de ale.act(a), es decir 0 o 1, lo cual significa que a reward se le puede sumar un número entre 0 y 4. Esto es porque como el juego es muy rápido se saltan un número de frames entre 2 y 4 y se le asignan las recompensas que se obtuvieron aleatoriamente para un frame
14. se agrega al final del vector rewards el valor del reward calculado
15. se suma al escalar totalreward el reward calculado
16. Si ale.game_over:
    1. Se incrementa el número de episodios
    2. Se resetea el juego
    3. Se obtiene la pantalla y se guarda en data
    4. Se ejecuta discount_rewards al vector de rewards con gamma =.99:
    5. Se ejecuta modulateGradient (dlogps,rewards): todos los elementos en el vector dlogps se multiplican por los respectivos rewards (ambos vectores son del mismo tamaño)
    6. Se ejecuta policy backward (se actualizan dw1 y dw2):
       - Se multiplica dw2=hiddenstates*dlogps (HxNumberOfGames) (cada juego tiene su reward)
       - Se multiplica dh=w2*dlogps (HxNumberOfGames)
       - Se hace prelu a dh con hiddenstates: para todos los elementos de dh, si su respectivo hiddenstate no es mayor que cero (es menor o igual a cero) dh es cero
       - Se multiplica dw1=dh*exs (HxD)
    7. Se ejecuta acum, a dw1 y dw2, es decir que se suma dw1 y dw2 a sus respectivos buffers component-wise
    8. Si se trata del episodio final del batch, es decir es un múltiplo de 10:
       - se hace una copia de dw1Buffer
       - Se multiplica el vector rmsPropW1Cache por el escalar DECAY_RATE (0.99)
       - Se multiplica component-wise y se actualiza dw1BufferCopy=dw1BufferCopy^2*(1-DECAY_RATE)
       - Se suma a rmsPropW1Cache dw1BufferCopy
       - para todos los elementos de dw1Buffer: (HxD) w1=w1+(LEARNING_RATE*dw1Buffer/(sqrt(rmsPropW1C
       - Se hace lo mismo para dw2Buffer (desde la copia de dw1Buffer)
       - llena con 0 dw1Buffer y dw2Buffer
       - limpia los vectores rewards, exs, hiddenstates y dlogps
       - si runningsreward es distinto de cero, runningreward=totalreward si no, runningReward = runningReward * 0.99 + totalReward * 0.01 (totalreward es el reward del episodio, runningreward es el rewardmean)

Bibliografía

[1] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. CoRR, abs/1606.04838, 2016.

[2] L. Bottou, O. Bousquet, The tradeoffs of large scale learning. In Proc. Advances in Neural Information Processing Systems 20 161–168 (2007).

[3] Y. A. LeCun, L. Bottou, G. B. Orr, K.R. Müller. Efficient backprop. Neural networks: Tricks of the trade, pages 9–48. Springer, 1998

[4] M.G. Bellemare, Y. Naddaf, J. Veness, M. Bowling, The Arcade Learning Environment: An Evaluation

Plataform for General Agents, Journal of Artificial Intelligence Research, vol. 47, pp 253-279, 2013
[5] Deep Reinforcement Learning: Pong from Pixles [en línea][Consulta: 28 mayo 2017]. Disponible en: http://karpathy.github.io/2016/05/31/rl/