

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 píxeles que constituyen el juego. Para lograr esto se propone el uso de una red neuronal. La entrada de la red sería la matriz de píxeles 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 $\sigma(x) = \frac{1}{(1+e^{-x})}$ y la función hinge $h(x) = \max\{0, x\}$ conocida como ReLU son funciones populares para utilizarse como función de activación s .

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

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

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 función dada $h(\cdot, \cdot) : \mathbb{R}^{d_x} \times \mathbb{R}^{d_y} \rightarrow \mathbb{R}^{d_y}$ consideramos la familia de funciones de predicción

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

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 $l : \mathbb{R}^{d_y} \times \mathbb{R}^{d_y} \rightarrow \mathbb{R}$ donde dado el par entrada-salida (x, y) produce la pérdida $l(h(x; w), y)$ cuando $h(x; w)$ y y son salidas predichas 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 $P : \mathbb{R}^{d_x} \times \mathbb{R}^{d_y} \rightarrow [0, 1]$ 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 $R : \mathbb{R}^d \rightarrow \mathbb{R}$ 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 $R_n : \mathbb{R}^d \rightarrow \mathbb{R}$ 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

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

donde para todo $k \in \mathbb{N} := \{1, 2, \dots\}$, 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 $\{1, \dots, n\}$ y α_k es un paso positivo. Cada iteración de este método involucra

sólo el cálculo del gradiente $\nabla f_{i_k}(w_{i_k})$ correspondiente a una muestra. A diferencia de un algoritmo de optimización determinístico, 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

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

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 inicial w_1
2. Para $k = 1, 2, \dots$ 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 bien:

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

de otra forma:

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

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 η 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 aprendizaje 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 política actual. Una forma de aprender una política de aproximación es maximizando la recompensa esperada usando métodos basados en el gradiente, por lo que en política gradient el gradiente de la recompensa esperada respecto a los parámetros de la política se puede obtener por medio del teorema de policy gradient para políticas estocásticas.

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

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

Si intentamos aprender la política, representamos la política 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 política tiene varianza muy grande por lo que puede tomar mucho tiempo el entrenamiento y se puede llegar a un óptimo local. Para aprender una política 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. Al igual que en un método Monte-Carlo, para aprender una política 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 significa 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 política para estimar la función value. Esto se conoce como actor-critic, donde el actor es la política 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 política de π
- Realizar una acción basada en la política 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 expectativa 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 política, 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 expectativa o esperanza:

$$\nabla_{\theta} E_x[f(x)] = \nabla_{\theta} \sum_x p(x) f(x) = \sum_x p(x) \nabla_{\theta} \log p(x) f(x) = E_x[f(x) \nabla_{\theta} \log p(x)]$$

Esto quiere decir que la forma de cambiar los parámetros de la política 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 del aprendizaje por refuerzo. 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.

Para usar el emulador ALE, es necesario descargar un ROM del juego en cuestión. Adicionalmente, es necesario elegir una interfaz para interactuar con el *framework*, ya que es posible controlarlo por medio de wrappers de python, mensajes o usando la interfaz de C++ directamente.

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.

Aprendiendo a jugar Pong

Se creó 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 210×160 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 da una recompensa por cada decisión, la cual puede ser +1 si la pelota pasa la barra del oponente, -1 si la pelota pasa la barra del agente 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 210×160 y da como salida la probabilidad de moverse hacia arriba.

Preprocesamiento

Para empezar es necesario preprocesar los datos, cada frame del juego es transformado tomando un recorte de la pantalla y un muestreo del resultado. Con esto reducimos el espacio de entrada a una matriz de 80×80 :

$$210 \times 160 \times 3 \rightarrow 80 \times 80$$

El código del preprocesamiento se muestra continuación:

```
std::vector<double> preprocessloop(std::vector<unsigned char> data, int width, int height) {
    int top = 34;
    int bottom = 194;
    std::vector<unsigned char> interestingData =
        std::vector<unsigned char>(data.begin() + top * width, data.begin() + bottom * width);
    std::vector<double> frame(80*80);
    int count = 0;
    for(int i = 0; i < COLS * 2; i++) {
        for(int j = 0; j < ROWS * 2; j++) {
            int value = (int)interestingData[i * ROWS * 2 + j];
            if(i % 2 == 0 && j % 2 == 0) {
                if(value == 87) {
                    frame[i/2 * ROWS + j/2] = 0.0;
                } else {
                    frame[i/2 * ROWS + j/2] = 1.0;
                }
            }
        }
    }
    return frame;
}
```

Adicionalmente, para capturar la noción de movimiento, se encuentra la diferencia entre *frames* consecutivos para obtener la matriz con la que se alimentará la red.

Cálculo de la acción

Para elegir la acción usamos el *forward pass* de la red neuronal, cuya arquitectura consta de dos capas, la capa oculta, con 200 neuronas y una capa de activación sigmoide:

$$f : \mathbb{R}^{6400} \rightarrow \mathbb{R}^{200} \quad (1)$$

$$h : \mathbb{R}^{200} \rightarrow \mathbb{R}^{+200} \quad (2)$$

$$g : \mathbb{R}^{+200} \rightarrow \mathbb{R} \quad (3)$$

$$\phi : \mathbb{R}^+ \rightarrow [0, 1] \quad (4)$$

De modo que nuestra red neuronal es una función que asigna una probabilidad (la probabilidad de moverse hacia arriba) a cada *frame* del juego:

$$f \circ g \circ h \circ \phi : \mathbb{R}^{6400} \rightarrow [0, 1] \quad (5)$$

Cada iteración que será muestreada de esta distribución para obtener el movimiento realizado por el agente. Si los pesos de la red fueran inicializados en 0 la red tardaría mucho tiempo en comenzar a tomar decisiones distintas para cada *frame*. Para evitar este comportamiento, inicializamos aleatoriamente las matrices de pesos con valores de una distribución normal estandar. Cabe mencionar que *C++* no cuenta con una implementación de la distribución normal por lo que usamos el método Box Muller:

```
double randNormal(double mean, double stddev) {
    static double n2 = 0.0;
    static int n2_cached = 0;
    if (!n2_cached)
    {
        double x, y, r;
        do
        {
            x = 2.0*rand()/RAND_MAX - 1;
            y = 2.0*rand()/RAND_MAX - 1;
            r = x*x + y*y;
        }
        while (r == 0.0 || r > 1.0);
        {
            double d = sqrt(-2.0*log(r)/r);
            double n1 = x*d;
            n2 = y*d;
            double result = n1*stddev + mean;
            n2_cached = 1;
            return result;
        }
    }
    else
    {
        n2_cached = 0;
        return n2*stddev + mean;
    }
}
```

La función sigmoide 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.

```
double policy_forward(std::vector<double> x, std::vector<double> w1, std::vector<double> w2, std::vector<double> hiddenStates) {
    double prob;
    std::vector<double> h= matrixMatrixMultiplication(w1.data(), x.data(), H, D, 1);
    int size = h.size();
    relu(h);
    hiddenStates.reserve(hiddenStates.size() + h.size());
    hiddenStates.insert(hiddenStates.end(), h.begin(), h.end());
    std::vector<double> logp= matrixMatrixMultiplication(w2.data(), h.data(), 1, H, 1);
    prob = sigmoid(logp[0]);
    return prob;
}
```


}

Una vez que hemos calculado la probabilidad que nuestra red nos da para elegir la acción, lanzamos un volado con esta probabilidad y elegimos la acción que el agente debe tomar. Si nos detemos a pensar un momento, el agente no sabe si ha elegido la acción correcta, y la recompensa no vendrá sino hasta que el juego termine, ¿como es posible entonces incentivar aquellas acciones buenas y desincentivar aquellas malas si no sabemos el resultado hasta tiempo despues? A este problema se le conoce como *credit assignment problem* ya que sabemos cual de todas las acciones del agente contribuyeron a la obtención de una recompensa positiva.

Para aliviar este problema lo que se hace es asumir que la etiqueta elegida es la correcta y dejar que el agente juegue cierto número de juegos y decidiendo al final de cada juego si los movimientos elegidos fueron buenos o malos. Para que esto funcione, ponderamos las decisiones tomadas dando un mayor peso a las decisiones tomadas más recientemente. 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.

```
void discount_rewards(std::vector<double> &rewards) {
    int size = rewards.size();
    double acum = 0.0;
    double meanAcum = 0.0;
    for(int i = size - 1; i >= 0; i--) {
        if(rewards[i] > 0.0 || rewards[i] < 0.0) {
            acum = 0.0;
        }
        acum = acum * GAMMA + rewards[i];
        rewards[i] = acum;
        meanAcum = meanAcum + rewards[i];
    }
    meanAcum = meanAcum / size;
    double stdAcum = 0.0;
    for(int i = 0; i < size; i++) {
        stdAcum = stdAcum + (rewards[i] - meanAcum) * (rewards[i] - meanAcum);
    }
    stdAcum = sqrt(stdAcum / size);
    for(int i = 0; i < size; i++) {
        rewards[i] = (rewards[i] - meanAcum) / stdAcum;
    }
}
```

Cuando derivamos la función sigmoide, el resultado se puede expresar en términos de la función original:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (6)$$

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} \quad (7)$$

$$= \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) \quad (8)$$

$$= (1 - \sigma(x))\sigma(x) \quad (9)$$

Esto simplifica el cálculo del gradiente para la función de pérdida:

$$\frac{\partial L}{\partial x} = y \frac{(1 - \sigma(x))\sigma(x)}{\sigma(x)} - (1 - y) \frac{(1 - \sigma(x))\sigma(x)}{1 - \sigma(x)} \quad (10)$$

$$= y(1 - \sigma(x)) - (1 - y)\sigma(x) \quad (11)$$

$$= y - \sigma(x) \quad (12)$$

Simulamos varios episodios siguiendo este método y guardando en cada momento la información sobre las decisiones, los *frames*, los gradientes y las recompensas obtenidas. Después de 10 episodios, acumulamos el gradiente y hacemos backpropagation sobre la matriz de pesos. Es aquí donde modificamos el comportamiento de la red y hacemos que cada vez ofrezca mejores resultados.

Multiplicación de matrices en CUDA

Tanto en el algoritmo de policy forward, como en el de policy backward se tienen que hacer multiplicaciones matriz por matriz de diferentes dimensiones, en otras ocasiones fueron multiplicaciones matriz vector. Estas operaciones del programa son las que se paralelizaron en CUDA.

Primero se programó la multiplicación de una matriz por un vector haciendo operaciones *gaxpy row oriented*, sin embargo para evitar una *race condition* se realizaron primero las multiplicaciones $r_{ij} * x_j$ en un thread cada una de un bloque, por lo que se hicieron un número de bloques igual a los renglones de la matriz. En cada bloque se hizo un punto de sincronización para después hacer la suma de las multiplicaciones y asignarlas a y . El kernel es el siguiente:

```
__global__ void gaxpy(double *y, double *a, double *x, int m, int n){
    int bid = blockIdx.x;
    int tid = threadIdx.x;
    extern __shared__ double dots_s[];
    if(bid<m)
        if(tid<n){

            dots_s[bid*n+tid] = a[bid*n+tid] * *(x+tid);
            __syncthreads();
            if(tid == 0){
                for(int i=1;i<n;i++){
                    dots_s[bid*n] +=dots_s[bid*n+i];
                }
                *(y+bid)=dots_s[bid*n];
            }
        }
}
```

Después se vio la necesidad de hacer la multiplicación matriz por matriz, por lo que se hizo con base en la multiplicación matriz-vector por medio de operaciones *saxpy*. En este caso también para evitar una *race condition* al actualizar el valor de $c_{i,j}$, se paralelizaron las multiplicaciones $a_{i,k} * b_{k,j}$. El kernel es el siguiente.

```
__global__ void gaxpymm(double *y, double *a, double *b, int m, int n, int p){
    int bid = blockIdx.x;
    int tid = threadIdx.x;
    extern __shared__ double dots_s[];
    if(bid<m)
        if(tid<n){
            for(int c=0;c<p;c++)
                dots_s[bid*n*p+tid*p+c] = a[bid*n+tid] * *(b+(tid*p+c));
            __syncthreads();
        }
}
```

```

    if(tid == 0){
        for(int c=0;c<p;c++){
            for(int i=1;i<n;i++){
                dots_s[bid*n*p+c] +=dots_s[bid*n*p+i*p+c];
            }
            for(int c=0;c<p;c++){
                *(y+(bid*p+c))=dots_s[bid*n*p+c];
            }
        }
    }
}

```

Backpropagation

Durante la etapa de *backpropagation* usamos un método del curso de Coursera llamado RSMprop.

En resumen nuestra función de pérdida es $\sum_i A_i \log p(y_i | x_i)$ 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.

Resultados

Para el ejercicio se entrenó la arquitectura descrita durante 30,000 episodios con un *learning rate* de 10^{-3} . El agente lentamente va aprendiendo estrategias que eventualmente le llevan a dominar a la máquina. Al momento de parar el entrenamiento, el aprendizaje no se había detenido, en promedio al finalizar los 30,000 episodios, el agente ganaba por 10 puntos a la inteligencia artificial que el juego trae por defecto. No sabemos si existe un límite en lo que la red puede aprender o si es posible que en algún punto el agente domine totalmente el juego. En este hipotético caso, el agente ganaría con un puntaje perfecto cada juego, es decir todos los episodios terminarían 21-0.

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) $w1=w1+(LEARNING_RATE*dw1Buffer/(\sqrt{rmsPropW1Cache}))$

- 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)

Conclusiones

Para el problema en cuestión se aprovechó la tarjeta gráfica para realizar las secciones del proceso que consistían de multiplicaciones matriciales. En general no vimos que el desempeño de la tarjeta realmente mejore el tiempo de entrenamiento para obtener resultados. Creemos que esto ocurrió por dos motivos. En primer lugar, el verdadero cuello de botella del algoritmo se encuentra en la implementación de la ALE, ya que en cada momento el simulador tiene que calcular el siguiente frame y no hay forma de darle la vuelta a esto. Por otro lado, la multiplicación ocurre relativamente poco, y la forma en la que accedamos a los elementos de cada *frame* no parece ser óptima. Sin embargo, fue positivo tener la experiencia de optimizar secciones puntuales de un proceso que funcionaba correctamente. En el futuro, nos gustaría implementar estas ideas en un juego más complicado, usando una función *softmax* en vez de una sigmoide para permitir la elección de un número no binario de acciones. En un principio, este era el objetivo del proyecto, pero nunca logramos que el agente aprendiera más allá del comportamiento aleatorio.

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 Platform for General Agents, Journal of Artificial Intelligence Research, vol. 47, pp 253-279, 2013
- [5] Deep Reinforcement Learning: Pong from Pixels [en línea][Consulta: 28 mayo 2017]. Disponible en: <http://karpathy.github.io/2016/05/31/rl/>
- [6] Overview of mini-batch gradient descent [en línea][Consulta: 30 mayo 2017]. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf