Trabajo Final

Programación Eficiente



Alumno: Arias Fernando Agustín

Legajo: 81723

Profesor: Maximiliano Andrés Eschoyez

Fecha de Presentación: 01/08/2024

Introducción 3

Programa 3

Análisis del Programa 3

Optimizaciones Realizadas 4

Pruebas 4

Pruebas de Memoria 5

Pruebas de Caché 6

Pruebas de Tiempo y Performance 7

Conclusión 8

Introducción

En este informe se detalla el proceso de análisis y optimización correspondiente a un programa elegido para la materia programación eficiente, aplicándole modificaciones a ese programa para ver si mejora su eficiencia, en cuanto a velocidad de ejecución, memoria utilizada, etc.

Programa

El programa en cuestión elegido es la implementación de una Red Neuronal llamada Perceptrón Multicapa (MLP, por sus siglas en inglés “Multi-Layer Perceptrón), utilizando una forma simplificada de back-propagation (retro propagación) y el descenso del gradiente estocástico(SGD).

Análisis del programa

Esta red consta de una capa de neuronas de entrada, una capa oculta, y una de salida.

Estructura de las capas:

* Capa de Entrada: En el código se utilizan 10 neuronas (“INPUT\_NODES”) que corresponden a las entradas del conjunto de datos.
* Capa Oculta: En el código se utilizan 10 neuronas (“HIDDEN\_NODES”) que son responsables de capturar patrones en los datos.
* Capa de Salida: En el código se utilizan 10 neuronas(“OUTPUT\_NODES”) que generan las predicciones de la red.

Función de Activación (define el valor de la salida de un nodo dada una entrada o un conjunto de entradas), tenemos 2:

* Función Sigmoide (“sigmoid”): Se utiliza como función de activación en las neuronas mapeando cualquier valor real entre un rango de 0 a 1.
* Derivada de la función Sigmoide: Se usa para calcular el error (“gradiente”: dirección) durante el algoritmo de retro propagación.

Inicialización de Pesos:

* Los pesos se inicializan con valores aleatorios entre 0 y 1, para que todas las neuronas aprendan características diferentes. Entonces una neurona de entrada tiene un dato de entrada y un peso.

El algoritmo utilizado se llama back – propagation con un proceso estocástico de cálculo de error similar al descenso del gradiente(SGD).

Se realiza en 3 pasos:

Forward Pass (propagación hacia adelante):

* Calculo de las Salidas de la Capa Oculta: Los datos de entrada se multiplican por los pesos de entrada y a ese resultado de la multiplicación se le va realizando una sumatorio hasta que se procesan todas las neuronas y se le aplica la función de activación para obtener los datos de salida de la capa oculta.
* Calculo de las Salidas de la Capa de Salida: Las salidas de la capa oculta se multiplican por los pesos de las neuronas de la capa oculta, se le realiza la sumatoria hasta procesar cada neurona y luego a la sumatoria se le aplica la función de activación para obtener los datos de la capa de salida.

Backward Pass (Retro propagación):

* Errores de la capa de salida: Se calcula la diferencia entre los datos de salida ya calculados con los objetivos(targets).
* Errores de la capa oculta: Se propaga el error hacia atrás desde la capa de salida a la capa oculta ajustando los pesos en función del “gradiente” del error y la derivada de la función de activación.

Como último paso se actualizan los pesos usando el “gradiente descendente” que sería la tasa de aprendizaje (que determina el tamaño de los pasos) multiplicando el error de salida y el vector de las neuronas de salida para saber los pesos de la capa oculta. Y para los pesos de la capa de entrada actualizados, multiplica el error de la capa oculta por la taza de aprendizaje y el vector de los datos de entrada.

Debido a la cantidad de cálculos matemáticos que hay que hacer en el programa original se utiliza el bucle for para recorrer las neuronas.

Y a medida que necesitamos recorrer los vectores y las matrices para actualizar los números y calcular los errores, los bucles for van incrementando. Lo cual termina siendo muy ineficiente la predicción de la red neuronal cuando se quiere trabajar con datos muy grandes.

Optimizaciones realizadas

En el nuevo Código optimizado se utilizó la biblioteca Eigen de C++ para realizar las operaciones con matrices y vectores.

* Permite realizar la multiplicación de matrices de manera más eficiente en comparación con las implementaciones de forma manual.
* Al terminar con los bucles explícitos (los que se necesitan para rellenar la matriz) Eigen utiliza funciones ya integradas para cargar los datos de las matrices y de los vectores de una forma más eficiente y rápida.
* Algunas funciones se realizan los cálculos en una función vectorizada en vez de iterar manualmente los vectores y las matrices.

Lo cual para realizar operaciones con matrices y vectores la biblioteca Eigen de c++ es muy versátil, y más si se trabaja con redes neuronales parecido a cuando usaríamos la biblioteca numpy para trabajar con datos en Python.

Pruebas

Las pruebas realizadas se basaron en 10 neuronas, para cada capa, una capa de entrada, una oculta, y una de salida.

Con una taza de aprendizaje muy chica: 0.00001: mientras más chica sea la distancia de actualización de pesos más precisa será la red.

Dado una matriz de objetivos con valores entre 0 y 1, el vector de predicción resultante es un conjunto de 10 datos representando a las 10 neuronas de salida con valores similares entre los dados en la matriz para tratar de estar en la misma región y acercarse lo más que se pueda.

Con lo cual para las pruebas se utilizaron herramientas vistas en clase

Pruebas de Memoria:

Programa Original:

==24100== HEAP SUMMARY:

==24100== in use at exit: 0 bytes in 0 blocks

==24100== total heap usage: 400,233 allocs, 400,233 frees, 32,096,656 bytes allocated

Programa Optimizado:

==25270== HEAP SUMMARY:

==25270== in use at exit: 0 bytes in 0 blocks

==25270== total heap usage: 1,100,014 allocs, 1,100,014 frees, 232,092,896 bytes allocated

Ambos resultados parecen indicar que no hay perdida de memoria en el programa. Sin embargo el Segundo resultado muetra un mayor uso de memoria en comparación con el primer resultado.

Es importante que el aumento de uso de la memoria se debe a las modificaciones implementadas para realizar el profiling.

Pruebas de Cache

* Programa Original:

==1927== I refs: 16,089,580,254

==1927== I1 misses: 2,445

==1927== LLi misses: 2,198

==1927== I1 miss rate: 0.00%

==1927== LLi miss rate: 0.00%

==1927==

==1927== D refs: 9,909,960,283 (6,564,148,908 rd + 3,345,811,375 wr)

==1927== D1 misses: 216,110 ( 212,631 rd + 3,479 wr)

==1927== LLd misses: 10,716 (8,349 rd + 2,367 wr)

==1927== D1 miss rate: 0.0% ( 0.0% + 0.0% )

==1927== LLd miss rate: 0.0% (0.0% + 0.0%)

==1927==

==1927== LL refs: 218,555 (215,076 rd + 3,479 wr)

==1927== LL misses: 12,914 ( 10,547 rd + 2,367 wr)

==1927== LL miss rate: 0.0% ( 0.0% + 0.0% )

* Programa Optimizado:

==3755== I refs: 15,263,432,797

==3755== I1 misses: 82,709,181

==3755== LLi misses: 3,102

==3755== I1 miss rate: 0.54%

==3755== LLi miss rate: 0.00%

==3755==

==3755== D refs: 10,104,224,728 (5,533,942,616 rd + 4,570,282,112 wr)

==3755== D1 misses: 291,456 ( 288,138 rd + 3,318 wr)

==3755== LLd misses: 10,089 ( 8,298 rd + 1,791 wr)

==3755== D1 miss rate: 0.0% ( 0.0% + 0.0% )

==3755== LLd miss rate: 0.0% ( 0.0% + 0.0% )

==3755==

==3755== LL refs: 83,000,637 ( 82,997,319 rd + 3,318 wr)

==3755== LL misses: 13,191 ( 11,400 rd + 1,791 wr)

==3755== LL miss rate: 0.0% ( 0.0% + 0.0% )

Las conclusiones que podemos sacar viendo los dos programas es que, el programa optimizado al utilizar más memoria tiene mayor porcentaje a que haya un miss rate de su memoria cache, sin embargo la diferencia no es grande es favorable, por ejemplo la cantidad de LL refs a comparación de la cantidad de LL misses apenas es mayor a la cantidad de LL misses en el programa original, y la cantidad de LL refs en el programa original son muchísimo más bajo que en el optimizado.

Pruebas de Tiempo y Performance

En la siguiente prueba se analizará el porcentaje de tiempo de cada parte del programa y los segundos que le lleva al procesador realizarla, colocaremos los resultados más relevantes ya que se analiza todo el programa y en las funciones y partes del programa que no utilizan muchos recursos el porcentaje de uso es cercano a 0, lo cual es despreciable.

Para la prueba se utilizó como datos de entrada 10 neuronas tanto de entrada, capa oculta, y de salida. Los números de ejemplos de entrada fueron 100, y la cantidad de iteraciones para que aprenda la red neuronal fueron 1000 iteraciones.

* Programa Original:

% acumulative self self total

time seconds seconds calls s/call s/call name

42.86 36.56 36.56 1 36.56 85.23

Función train de NeuralNetwork (almacenando en memoria sus parámetros)

25.46 58.28 21.72 2010085938 0.00 0.00

Declaración y almacenamiento de matrices

12.80 69.20 10.92 705053104 0.00 0.00

Declaración y almacenamiento de vectores

4.96 73.43 4.23 2100000000 0.00 0.00

Declaración y llamada del vector const

4.65 77.40 3.97 2100000100 0.00 0.00

Declaración y llamada del otro vector const

2.36 79.42 2.02 200000020 0.00 0.00

Función Sigmoide

1.34 80.56 1.15 100000000 0.00 0.00

Función Derivada de la Sigmoide

Y continua con las llamadas a los vectores y almacenándolas en memoria el % de tiempo ya empieza a acercarse al 0 , pero los segundos acumulables de la cantidad de llamadas al ejecutar el programa llegan a un máximo de: 85.30s.

* Programa Optimizado

% cumulative self self total

time seconds seconds calls s/call s/call name

3.25 0.14 0.14 30000200 0.00 0.00

Eigen: conj\_helper se relaciona con operaciones de conjugación de un número.

3.25 0.28 0.14 200002 0.00 0.00

Eigen: general\_matrix\_vector\_product: estructura para manejar el cálculo del producto de una matriz general con un vector.

3.02 0.41 0.13 20000000 0.00 0.00

Eigen: binary\_evaluator: cwiseBinary se prepara para manejar operaciones de dos operando

Como se diferencia en los 3 primeros resultados, la distribución de la memoria esta pareja con cada función que ayuda la biblioteca Eigen para calcular matrices y vectores.

A diferencia del programa original, la cantidad de segundos que le toma a cada función del programa tarda mucho menos, llegando a un máximo de 4.31 segundos.

Conclusión

Se puede observar la importancia de aplicar técnicas de optimización para mejorar la performance del programa. Las librerías que existen a veces nos ayudan a simplificar el problema y tener una mejoría en la estructura de datos y optimización de uso de bucles. Estos cambios pueden significar una mejoría de velocidad y eficiencia del programa.