DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Final

30 de julio de 2017 — Organización del Computador II

Integrante	LU	Correo electrónico
Costa, Manuel	COMPLETAR	COMPLETAR
Gatti, Mathias	477/14	mathigatti@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción **3**

2. Manual de Usuario **4**

2.1. Requerimientos 4

2.2. Preparación de datos de entrenamiento 4

2.3. Ejecución del programa 4

3. Tópicos Teóricos **6**

4. Implementación **7**

4.1. Datos de Entrada 7

4.2. Secciones Principales 7

4.3. Métodos implementados Assembler 7

4.3.1. cost_derivative 8

4.3.2. vector_sum 8

4.3.3. update_weight 8

4.3.4. hadamard_product 8

4.3.5. matrix_prod 9

4.4. Tests 9

5. Resultados **10**

5.1. Porcentaje de aciertos de las redes 10

5.2. Tiempos 10

6. Conclusiones **11**

1. Introducción

En el presente trabajo se realiza la implementación de una red neuronal sigmoidea con una capa oculta, usando los lenguajes C99 y Assembly x64. Debido a que dicho tipo de redes se basa fuertemente en operaciones vectoriales, encontramos en este proyecto una buena excusa para probar distintas optimizaciones usando SIMD.

Dicha red fue diseñada para resolver la clásica tarea de reconocimiento de digitos manuscritos, utilizando el dataset MNIST. Escogimos esta tarea por ser un problema de tamaño razonable para tratar con el hardware del que disponemos, pero que a su vez es lo suficientemente grande como para permitir un análisis interesante.

Otra de las razones para escoger este dataset es que el objetivo central que se persigue es el de conseguir una optimización desde el punto de vista temporal, por lo que no se busca profundizar en las distintas técnicas conocidas para mejorar la precisión de la red, sino que nos conformamos con una versión bastante simple de la misma. En este sentido, un dataset más complicado no nos aportaría nada *a priori*.

2. Manual de Usuario

En las próximos secciones se hablará en mayor detalle del funcionamiento del programa implementado pero para los usuarios que quieran utilizar rapidamente nuestro software sin necesidad de entrar en detalles técnicos se describe a continuación lo necesario para ejecutarlo y los pasos a seguir.

2.1. Requerimientos

Para utilizar nuestro programa se necesita tener instalado *Python*2,7 y *C*.

Las bibliotecas de Python utilizados son las siguientes:

- cPickle
- gzip
- numpy
- matplotlib
- subprocess

2.2. Preparación de datos de entrenamiento

Una vez cumplidos los requerimientos se debe ejecutar el script de *Python* ubicado en la carpeta *mnist* en la raiz del proyecto. Este creará los archivos de entrenamiento que utilizará nuestro programa en la carpeta *data*.

2.3. Ejecución del programa

Con los archivos de entrenamiento listos ya podemos ejecutar el programa, este se ejecuta a partir de *program.py* ubicado en la raiz del proyecto. Este programa recibe 3 parámetros, primero el lenguaje que queremos utilizar (C̈o äsm̈), luego el tipo de dato (f̈loatö d̈ouble) y por ultimo la ubicación de la imagen del digito que queremos predecir.

La imagen debe ser de 28x28 pixeles, como ejemplo esta $test_image.png$, para predecir el caracter escrito se utilizará entonces el siguiente comando.

```
1 python program.py asm float test_image.png
```

Al ejecutar este comando se debería recibir como resultado algo similar a lo que se ve en la siguiente imagen.

Figura 1: Imagen de ejemplo del output del programa

3. Tópicos Teóricos

–COMPLETAR CON TOPICOS TEORICOS–

4. Implementación

En esta sección se explican los detalles implementativos de nuestro clasificador.

4.1. Datos de Entrada

El programa tiene el set de datos de MNIST ubicado en la carpeta *mnist* en un archivo comprimido, este es convertido a archivos de texto que contienen un pixel por línea con un script de python el cual aloja los archivos en la carpeta *data*. Esto se hace para facilitar la posterior lectura de estos datos por nuestro programa.

Una vez hecho esto se puede utilizar program.py para compilar y ejecutar nuestro código de c que entrena una red neuronal para reconocer estos caracteres. Tanto en la carpeta $float$ como en la carpeta *double* se encontrará el código que implementa dicho programa y las correspondientes herramientas de compilación. Para compilar los archivos fuente manualmente basta con utilizar el `Makefile`, este crea los archivos `asm_version` y `c_version` que ejecutan la red neuronal.

4.2. Secciones Principales

En las carpetas *double* y $float$ hay versiones equivalentes del clasificador para estos dos tipos de datos.

Dentro de las carpetas hay 3 archivos con código fuente, *helpers*, *tensorOps* y *nn*.

Los archivos *helpers* implementan varias funciones útiles, entre otras cosas las que utilizamos para leer los TXTs donde estan los datos de entrenamiento, también algunas funciones matriciales básicas como transpocisión e impresión.

Luego esta *tensorOps* en el cual pusimos los métodos que implementamos tanto en C como en $ASSEMBLER$ estas son operaciones matriciales y vectoriales como el producto matricial y la suma vectorial.

Por último esta *nn* que es el archivo que utiliza a los demás para implementar toda la lógica de la red neuronal y donde se encuentra el main de nuestro programa.

4.3. Métodos implementados Assembler

A continuación damos una breve explicación de los métodos que implementamos en assembler para intentar mejorar la performance se describen a continuación. Todos ellos poseen la propiedad de realizar cierta operación aritmética repetidas veces para distintos valores haciendo propicio el intento de paralelización con SIMD.

4.3.1. cost_derivative

Este método es el gradiente del error cuadrático medio lo cual se reduce simplemente al computo de la resta entre vectores.

```
1  void cost_derivative(double* res_vec, double* target_mat, uint cant_imgs, double* output) {
2    for (int i = 0; i < 10; i++) {
3      for (uint j = 0; j < cant_imgs; j++){
4          output[i * cant_imgs + j] = res_vec[i * cant_imgs + j] - target_mat[i * cant_imgs + j];
5      }
6    }
7  }
```

4.3.2. vector_sum

Como indica su nombre esta método computa la suma vectoria.

```
1  void vector_sum(double* vector1, double* vector2, uint n, double* output){
2  // |vector1| == |vector2| == n
3    for (int i = 0; i < n; i++) {
4      output[i] = vector1[i] + vector2[i];
5    }
6  }
```

4.3.3. update_weight

COMPLETAR

```
1  void update_weight(double* w, double* nw, uint w_size, double c){
2    for(uint i = 0; i < w_size; i++){
3      w[i] -= c * nw[i];
4    }
5  }
```

4.3.4. hadamard_product

El producto de Hadamard consiste en la multiplicación componente a componente entre dos matrices.

```
1  void hadamard_product(double* matrix1, double* matrix2, uint n, uint m, double* output){
2  /* matrix1 and matrix2 are nxm*/
3    for(uint i = 0; i < n; i++){
4      for(uint j = 0; j < m; j++){
5        output[i * m + j] = matrix1[i * m + j] * matrix2[i * m + j];
6      }
7    }
8  }
```

4.3.5. matrix_prod

Este es el clásico producto matricial donde $matrix1$ es de $n \times m$, $matrix2$ es de $m \times l$ y la variable de salida, $output$, de dimensiones $n \times l$.

```
1  void matrix_prod(double* matrix1, double* matrix2, uint n, uint m, uint l, double* output){
2  // matrix1 is nxm
3  // matrix2 is mxl
4  // output is nxl
5    for(uint i = 0; i < n; i++) {
6      for(uint j = 0; j < l; j++){
7        output[i * l + j] = 0;
8        for(uint k = 0; k < m; k++){
9          output[i * l + j] += matrix1[i * m + k] * matrix2[k * l + j];
10       }
11     }
12   }
13 }
```

4.4. Tests

Tanto la versión del programa que recibe Floats como la que recibe Doubles tiene un conjunto de tests en las funciones que implementamos tanto en ASSEMBLER como C, esto fue para corroborar el buen funcionamiento de lo dessarrollado y ayudarnos a debuguear.

Dichos tests se encuentran en la carpeta *test* y se compilan con el `Makefile` el cual crea el ejecutable *test* que corre los mismos.

5. Resultados

5.1. Porcentaje de aciertos de las redes

La red neuronal tiene un porcentaje de acierto del 95 % en todas sus versiones. A pesar de que el tipo float tiene una menor capacidad de representación numérica que su contraparte el tipo double su precisión parece ser suficiente para realizar los cálculos necesarios y hacer que la red neuronal llegue a las mismas conclusiones que la red que utiliza double.

5.2. Tiempos

Aquí se puede observar un cuadro de resultados comparando nuestras implementaciones de ASSEMBLER contra las de C compiladas con gcc -03.

Versión	cost_derivative	mat_plus_vec	update_weight	hadamard_product	matrix_prod
Double C	0.019694	0.323289	0.298969	0.966852	0.841608
Double ASM	0.019881	0.338240	0.758107	0.341586	0.498035
Float C	0.019256	0.171635	0.158144	0.922496	0.711830
Float ASM	0.018904	0.170835	0.381136	0.167613	0.490826

Tabla 1 Cuadro de resultados obtenidos en cada método.

6. Conclusiones

–COMPLETAR CONCLUSION–