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	35/14	manucos94@gmail.com
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

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.

En las próximas secciones se explicará el trasfondo teórico de nuestro programa con el cual luego daremos una breve explicación de los detalles de la implementación que realizamos, llegando al final a los tiempos obtenidos donde intentaremos verificar si SIMD es realmente una buena alternativa para bajar los tiempos de computo de la red neuronal.

El resultado final será un programa perfectamente funcional y capaz de identificar dígitos numericos con una muy buena precisión y con una interfaz cómoda para su fácil utilización.

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

En esta sección esperamos poder brindar una breve introducción a los temas que son relevantes para el desarrollo y la implmentación de este proyecto. La dividiremos en tres partes: qué dataset elegimos y por qué; brindaremos una breve explicación que busca proveer al lector de las herramientas para poder interpretar la implementación; finalmente, explicaremos porque la aplicación de SIMD resulta pertinente a este problema.

3.1. El dataset: MNIST

El dataset MNIST[1] está compuesto por imágenes de dígitos manuscritos del 0 al 9, con una resolución de 28×28 píxeles. A su vez, ya viene dividido en un training set de 60000 ejemplos (en particular, nosotros solo usamos 50000), y un test set de 10000.

Es un dataset que, por su simplicidad, está pensado para ser usado como un primer benchmark rápido para modelos, pudiendo abstraerse de las complicaciones inherentes al preprocesamiento de datos.

En vista de que el objetivo central que se persigue es el de conseguir una optimización desde el punto de vista del tiempo de ejecución de las operaciones básicas, y no de la precisión del modelo (es decir, no nos interesa una red particularmente compleja), encontramos que este dataset se ajusta bien a nuestras necesidades: es lo bastante chico como para poder manejarlo con el hardware del que disponemos, pero no tanto como para no permitirnos hacer un análisis interesante. Un dataset más complicado no nos aportaría nada.

3.2. Redes Neuronales Artificiales

Daremos una explicación muy acotada al *scope* de este trabajo, de en qué consiste una Red Neuronal Artificial (RN).

Una RN es uno de los tantos modelos encuadrados dentro del paradigma del aprendizaje supervisado[2]. El elemento fundamental de las redes neuronales son las *neuronas*. Una neurona computa una función con múltiples inputs y un output (todos números reales). La imagen (2) ilustra la estructura general de una neurona.

La función que ejecuta la neurona tiene dos partes. Primero una lineal (también llamada *transfer function*), en la cual se multiplica a cada uno de los inputs x_i por un cierto peso w_i, y posteriormente se los suma. En la suma suele participar un término independiente, b. Es decir,

$$z = \sum w_i x_i + b$$

Luego, se le aplica a z la llamada *función de activación*, que nos dará el output de

[1] http://yann.lecun.com/exdb/mnist/
[2] https://en.wikipedia.org/wiki/Supervised_learning

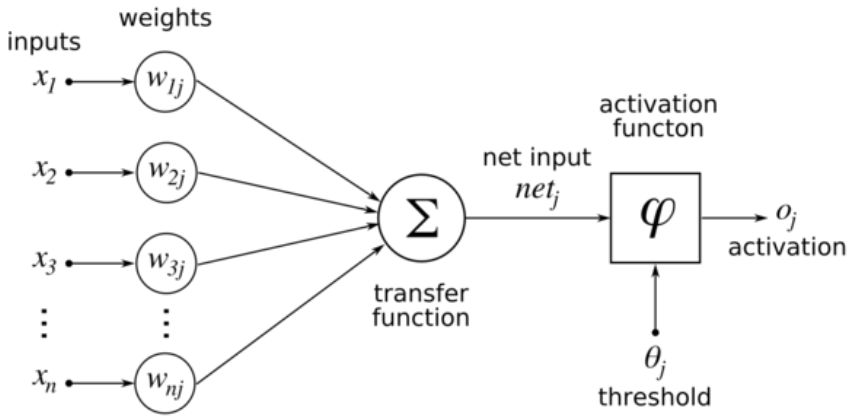

Figura 2: Estructura general de una neurona

la neurona. Dicho función puede ser cualquiera que vaya de los reales a los reales, aunque típicamente se escojen ciertas funciones no lineales (se puede probar fácilmente que usar una función lineal no agrega mayor capacidad para aproximar funciones). En particular, para este trabajo usaremos como función de activación la función sigmoidea definida como sigue:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

En general, una red neuronal va a consistir de muchas neuronas interconectadas (es decir que el output de una se vuelve el input de otra). Esto puede hacerse con diversas arquitecturas. En particular, nosotros usamos una de las más básicas que es la de Feedforward Neural Network. Esta es una arquitectura por capas o layers (donde cada capa es un conjunto de neuronas que no están interconectadas entre sí), en la cual el output de cada neurona de una capa alimenta al input de todas las neuronas de la capa siguiente (*full connected*). En la siguiente imagen se ilustra esta arquitectura

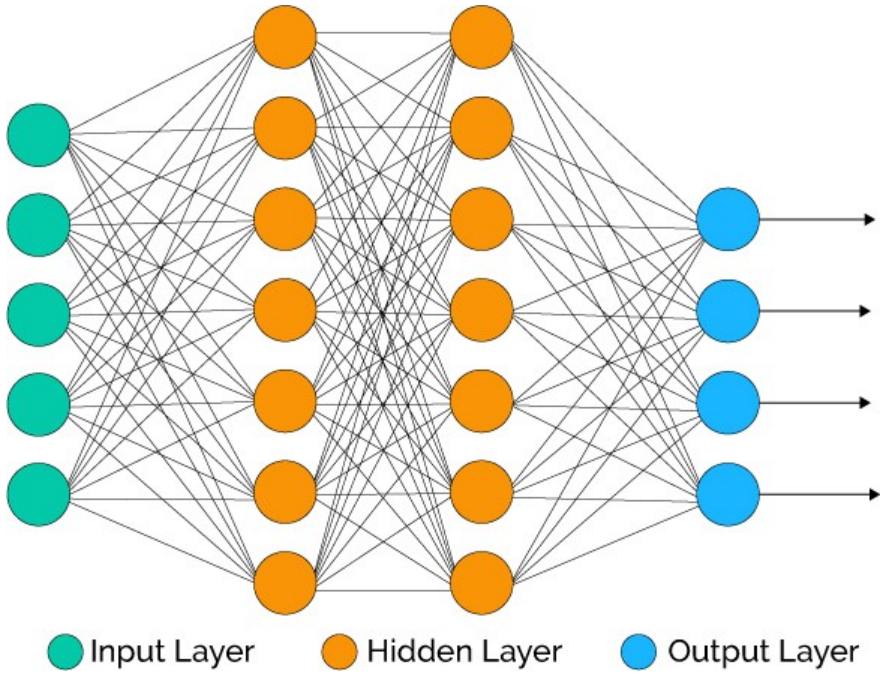

Figura 3: Estructura general de una feedforward neural network

Puede verse que se diferencian tres tipos de capas: input layers, hidden layers y output layers. La primera y la última son bastante autoexplicativas; las capas ocultas (hidden) reciben ese nombre por el hecho de que los cómputos que realizan no son visibles por el usuario de la red, en contraposición con los inputs y los outputs que sí son "visibles". A una red como la de la imagen se la llama 3-layer neural network (no se cuenta la capa de input).

Pasemos ahora a la parte más importante, que es el algoritmo de entrenamiento de la red. Ante todo es importante entender que lo que queremos aprender son los parámetros w_{ij} y b_i. Además, es necesario definir una función de costo: es decir, una función que nos indique cuan "lejos" están las predicciones de las etiquetas verdaderas. Luego, el objetivo del entrenamiento será seleccionar los parámetros w_{ij} y b_i que minimizen esta función.

Usaremos como función de costo el Error Cuadrático Medio (ECM)

$$C(w,b) = \frac{1}{2n}\sum_x ||y(x) - a||^2$$

donde w y b son todos los parámetros de nuestra red, $y(x)$ es el output de la red para el input x, a es el target verdadero para el input x, y n es la cantidad de casos de entrenamiento.

La heurística de optimización usada es la versión más simple de Gradient Descent, por lo que en cada epoch ajustaremos los parámetros de la siguiente forma:

$$w_{ij} := w_{ij} - \eta \frac{\partial C}{\partial w_{ij}}$$

$$b_i := b_i - \eta \frac{\partial C}{\partial b_i}$$

donde η es el *learning rate* que determina cuanto queremos modificar nuestros parámetros en una iteración, y $\frac{\partial C}{\partial v}$ es la derivada parcial de C respecto del parámetro v.

La pregunta que queda entonces es cómo calculamos las derivadas parciales. Para esto se utiliza un algoritmo conocido como *backpropagation*. Recibe este nombre en contraposición al *forwardpropagation*, que es la pasada que se realiza sobre la red para calcular el output. Es decir que ahora querremos atravesar la red en sentido opuesto para calcular las derivadas parciales.

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. Estructura de la Red Neuronal

La estructura de la red implementada en los archivos nn esta dada por una capa de entrada formada por 784 inputs (28^2) las cuales representan cada uno de los pixels de las imágenes. Esta capa se conecta a la capa interna de nuestra red y luego de ahí se va a la capa externa, formada por 10 salidas representando cada uno de los posibles digitos resultantes.

Por ejemplo, al ingresar los valores de una imagen que represente al número 2 se espera-

rá que la tercera neurona de salida (Ya que se empieza a contar desde el cero) indique 1 y todas las demas outputs indiquen 0.

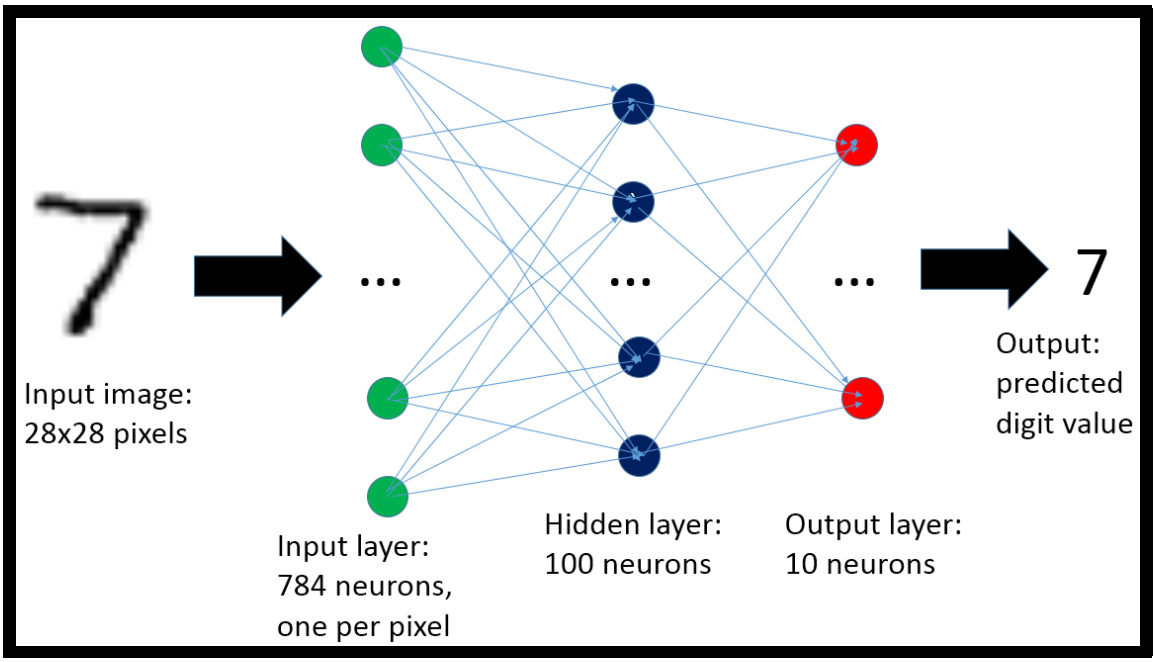

Figura 4: Diagrama de la red implementada

Es importante notar que si bien la cantidad de neuronas en la capa interna y externa esta fijada por el dominio del problema, la cantidad de neuronas en la capa oculta es una variable con la que se puede jugar, en la imagen se ve el caso en que se utilizan 100 neuronas como capa interna pero nosotros utilizamos 30 para la mayor parte de la experimentación ya que con este valor obtuvimos un buen trade-off entre tiempo de computo y performance.

4.4. 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.

4.4.1. 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   }
```

4.4.2. 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  }
```

4.4.3. 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  }
```

4.4.4. 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  }
```

4.4.5. 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 }
```

4.5. 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 promedio 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

A continuación se listan, para los distintos algoritmos que fueron optimizados, tablas que permiten comparar los tiempos promedios para las distintas versiones implementadas, junto con el desvío estándar en cada caso.

Dado que las optimizaciones implementadas en ASM diferencian los casos en que ciertas dimensiones de los parámetros son divisibles por 2 (si usamos Double) o 4 (si usamos Float), decidimos usar siempre dimensiones multiplo de 4. Esto es para poder ver el mejor caso de la optimización.

Los tiempos en esta sección están expresados en milisegundos salvo que se explicite lo contrario.

En total se corrieron 400k iteraciones de cada algoritmo para estimar estos valores, con un procesador i5-5300U y 8GB de memoria RAM. La versión en C fue compilada con la compilación más agresiva (O3).

cost_derivative Como parámetros se le pasaron dos matrices de tamaño 10×32. En este caso la dimensión que nos importa que sea múltiplo de 4 es la segunda. [TODO: completar comentando que esta es una funcion particularmente chica y fue facil de optimizar en ASM]

Versión	Media	Desvío estándar
Double C	0.251953	0.088764
Double ASM	0.128182	0.060222
Float C	0.252914	0.061287
Float ASM	0.083133	0.048002

vector_sum Esta función recibió como parámetros dos vectores de longitud 1000.

Versión	Media	Desvío estándar
Double C	0.344978	0.085860
Double ASM	0.415999	0.112268
Float C	0.207263	0.062842
Float ASM	0.229253	0.070555

update_weight Tomó como parámetros dos vectores de longitud 1000.

Versión	Media	Desvío estándar
Double C	0.392532	0.108260
Double ASM	0.393279	0.683633
Float C	0.242674	0.061426
Float ASM	0.223944	0.680904

hadamard_product Recibió dos matrices de dimensiones 1000×10. En este caso para que la optimización se aproveche al máximo el producto de las dimensiones debe ser divisible por 4.

Versión	Media	Desvío estándar
Double C	0.307460	0.163436
Double ASM	0.137181	0.062210
Float C	0.256564	0.080366
Float ASM	0.093358	0.053049

matrix_prod Se le pasaron dos matrices, una de dimensión 10×20 y la otra de 20×30. En este caso la dimensión que nos importa que sea multiplo de 4 es la dimensión en común entre ambas matrices.

Versión	Media	Desvío estándar
Double C	7.145342	1.293915
Double ASM	3.770623	1.011386
Float C	6.704427	0.491350
Float ASM	4.033694	0.324861

Finalmente mostramos el tiempo promedio que insume realizar un epoch[3]. Los hiperparámetros usados fueron:

- cantidad de unidades de la capa oculta = 30
- mini_batch = 32
- epochs = 50
- learning_rate = 3.0 (esto no afecta el tiempo que tarda un epoch)

Versión	Media (en segundos)
Double C	3.799580
Double ASM	3.054587
Float C	3.395475
Float ASM	2.576224

[3]Un epoch es una pasada de entrenamento sobre un mini-batch

6. Conclusiones

Este trabajo nos sirvió para aplicar y desarrollar el conocimiento adquirido en la materia a un tema que nos interesaba. Al finalizarlo pudimos realizar una red neuronal perfectamente funcional en C y $ASSEMBLER$ a un problema concreto e interesante con una performance superior a la que obtenian expertos en el área hace menos de una década[4], esto habría sido imposible sin cursar Organización del Computador 2. El trabajo desarrollado nos permitió aprender mas de los lenguajes de programación, herramientas de compilación, desarrollo de experimentos y demás temas que vimos en la materia.

Al finalizar este trabajo pudimos corroborar exitosamente la hipótesis de que la utilización de SIMD resultaría en una mejora en la performance temporal de nuestro programa en ciertas partes críticas. Viendo en detalle los resultados de nuestros experimentos pudimos ver como para algunos casos como $update_weight$ la implementación en $ASSEMBLER$ no valió la pena ya que sus tiempos fueron muy similares a los de su versión en C, como caso extremo tuvimos a $vector_sum$ el cuál mostro tiempos ligeramente mayores en $ASSEMBLER$ que en C, esto creemos que fue debido a que COMPLETAR.

Por otro lado métodos como $hadamard\_product$ y $matrix\_prod$ lograron tiempos considerablemente superiores con tiempos que duplicaron a sus versiones en C.

Cómo conclusión final entendemos que bajo ciertas circunstancias, donde mejoras en tiempo son cruciales, la utilización de SIMD puede ser una herramienta fundamental, aunque al mismo tiempo hay que tener en cuenta que los tiempos de desarrollo suelen ser mayores debido al bajo nivel de $ASSEMBLER$, incluso para funciones simples como las que hicimos y como vimos con $update_weight$ y $vector\_sum$ para algunos casos incluso pueden encontrarse resultados mediocres.

[4] http://yann.lecun.com/exdb/mnist/