DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Final

11 de diciembre 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 se hablará de los tiempos obtenidos donde intentaremos verificar si SIMD es realmente una buena alternativa para mejorar la performance temporal de la red neuronal.

El resultado final será un programa perfectamente funcional y capaz de identificar dígitos numéricos 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 rápidamente nuestro software sin necesidad de entrar en detalles técnicos se describen a continuación los pasos a seguir para ejecutarlo.

2.1. Requerimientos

Para utilizar nuestro programa se necesita tener instalado Python 2.7 y C.

Las bibliotecas de Python utilizadas 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, los cuales están ubicados 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 (*C* o *asm*), luego el tipo de dato (*float* o *double*) y por último la ubicación de la imagen del dígito que queremos predecir.

La imagen debe ser de 28x28 pixeles, como ejemplo está *test_image.png*, para predecir el carácter escrito se utilizará entonces el siguiente comando.

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

Al ejecutar este comando se debería preparar la versión de red neuronal que tiene métodos implementados en *ASSEMBLER* y que representa los valores de las matrices con floats.

Se debería recibir en la terminal 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 implementación 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. En este sentido, un dataset más complicado no nos aportaría nada.

3.2. Redes Neuronales Artificiales

No es la idea de esta sección brindar una introducción al amplio mundo de las RNAs. Nuestro objetivo es meramente dar definiciones mínimas y algoritmos necesarios (sin ninguna justificación teórica) para facilitar la interpretación del código. Para una explicación más profunda existe una basta bibliografía. En particular, este trabajo está fuertemente influenciado por el siguiente libro online: http://neuralnetworksanddeeplearning.com/.

3.2.1. Definiciones

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

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

peso w_i, y posteriormente se los suma. En la suma suele participar un término independiente (llamado *bias*), 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 la neurona. Dicha función puede ser cualquiera que vaya de los reales a los reales, aunque típicamente se escogen 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:

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

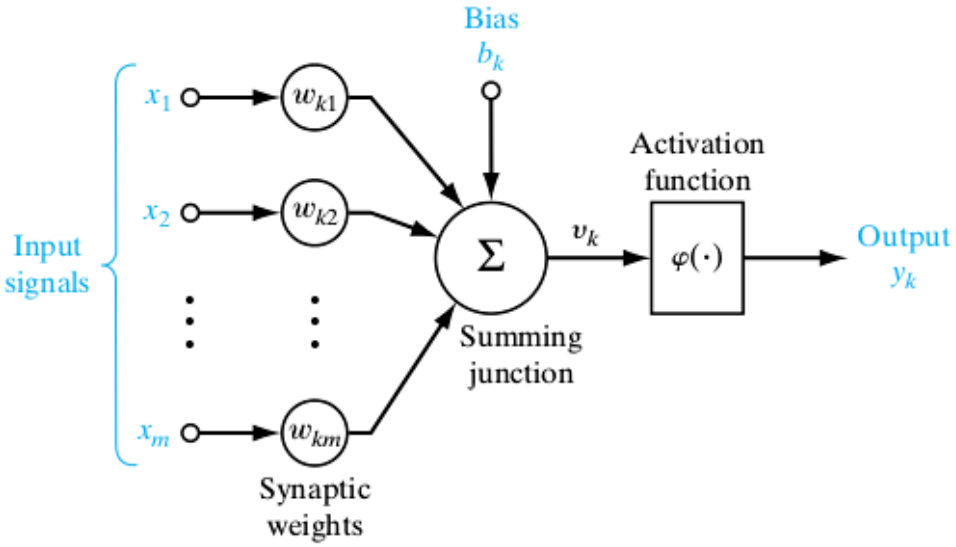

Figura 2: Estructura general de una neurona.

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

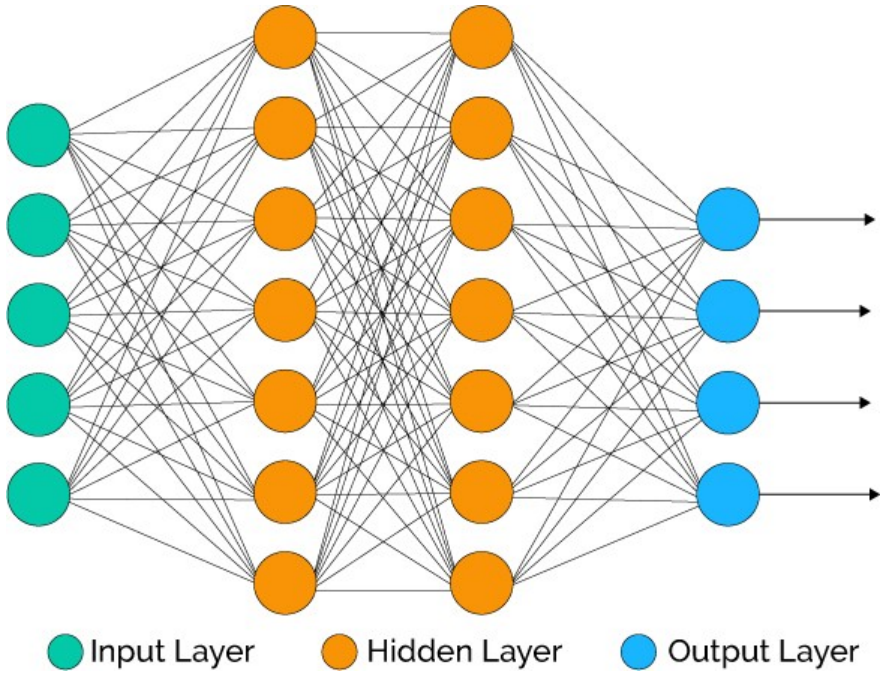

Figura 3: Estructura general de una feedforward neural network

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).

Por cuestiones que comentaremos en la sección siguiente, es conveniente adaptar una notación matricial para trabajar con los parámetros (pesos y *biases* de las neuronas). Notaremos w_{ij}^l como el peso correspondiente al input j-ésimo de la i-ésima neurona en la capa l. Similarmente b_i^l será el bias correspondiente a la neurona i-ésima de la capa l.

Entonces, podemos definir las matrices W^l tales que $(W^l)_{ij} = w_{ij}^l$, y los vectores b^l tales que $(b^l)_i = b_i^l$ (no nos volvemos a referir a los valores individuales, así que los paréntesis no se usan más).

Las ecuaciones vistas antes para el cálculo del output de una neurona pueden generalizarse para el output de una capa de la siguiente manera:

$$z^l = W^l a^{l-1} + b^l$$
$$a^l = \sigma(z^l)$$

donde $1 \leq l \leq L$, y definimos $a_0 = x$, con x el input de la red. σ en este caso es una función sobre un vector o una matriz y se aplica en forma *element-wise*. Algo que puede generar confusión es que x (y por lo tanto las subsecuentes a^l) puede ser tanto un vector como una matriz, de acuerdo a si hay uno o muchos inputs (cada input es un vector columna)

tratando de computarse en simultáneo. Como usamos $mini-batches$ para entrenar, siempre será una matriz durante la fase de entrenamiento.

3.2.2. Entrenamiento

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^l y b^l. Además, es necesario definir una función de costo: es decir, una función que nos indique cuán "lejos" están las predicciones de las etiquetas verdaderas. Luego, el objetivo del entrenamiento será seleccionar los parámetros W^l y b^l que minimicen 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 Gradient Descent[3][4], por lo que en cada epoch ajustaremos los parámetros de la siguiente forma:

$$W^l_{ij} := W^l_{ij} - \eta \frac{\partial C}{\partial W^l_{ij}}$$

$$b^l_i := b^l_i - \eta \frac{\partial C}{\partial b^l_i}$$

donde η es el *learning rate* que determina cuánto 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 *forward-propagation*, 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.

Nuevamente, no es la intención dar un *insight* teórico sobre los algoritmos. Nos limitaremos a presentar el mismo, una justificación puede encontrarse en el libro citado arriba.

Para una L-layer neural network (recordar que con la input layer en rigor son L+1 layers), queremos calcular las matrices $\nabla W^l = \frac{\partial C}{\partial W^l}$, y los vectores $\nabla b^l = \frac{\partial C}{\partial b^l}$. Notar que

[3] https://en.wikipedia.org/wiki/Gradient_descent

[4] En realidad, más correctamente usamos Stochastic Gradient Descent. Esto significa que en lugar de usar toda la data de entrenamiento antes de actualizar los parámetros, usamos pequeñas porciones (*mini-batches*) sampleadas aleatoriamente, lo que permite una mayor velocidad de convergencia. Debido a que es una técnica estándar del área decidimos aplicarla.

estamos cometiendo un abuso de notación donde la derivada parcial respecto a una matriz (o un vector) es la matriz (o vector) que tiene las derivadas respecto de cada una de las componentes originales.

Algorithm 1 Backpropagation

procedure BACKPROPAGATION(Ws, bs, X, y)
  $a^0 := X$ ▷ Comienza pasada forward
  for $l \leftarrow 1.... L$ **do**
    $z^l := W^l a^{l-1} + b^l$
    $a^l := \sigma(z^l)$
  $\delta^L := (a^L - y) \odot \sigma'(z^L)$ ▷ Comienza pasada backward
  $\nabla b^L := \delta^l$
  $\nabla W^L := \delta^l a^{(l-1)T}$
  for $l \leftarrow (L-1) \dots 1$ **do**
    $\delta^l := ((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$
    $\nabla b^l := \delta^l$
    $\nabla W^l := \delta^l a^{(l-1)T}$

3.3. Vectorización y Redes Neuronales

En la sección anterior dimos definiciones y algoritmos en términos de matrices y vectores, lo cual puede resultar un poco más engorroso que si hiciéramos los cómputos de cada componente individualmente.

La razón por la cual se decide hacer esto es una cuestión de performance: vectorizar permite explotar el paralelismo que dan las operaciones SIMD del procesador (o de una GPU en el caso más usado).

La naturaleza intrínsecamente vectorial de las redes neuronales fue lo que nos llevó a querer plantearnos este trabajo.

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 píxel 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 transposición e impresión.

Luego está *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. Es importante entender el objetivo de la implementación en $ASSEMBLER$, al utilizar SIMD podemos realizar 2 operaciones de *double* a la vez y 4 de $float$, por ejemplo la suma, multiplicación y reordenamiento de valores en un registro. Esta herramienta nos permitirá paralelizar los cálculos e idealmente mejorar los tiempos significativamente.

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 dígitos resultantes.

Por ejemplo, al ingresar los valores de una imagen que represente al número 2 se esperará que la tercera neurona de salida (Ya que se empieza a contar desde el cero) indique 1 y todas las demás outputs indiquen 0.

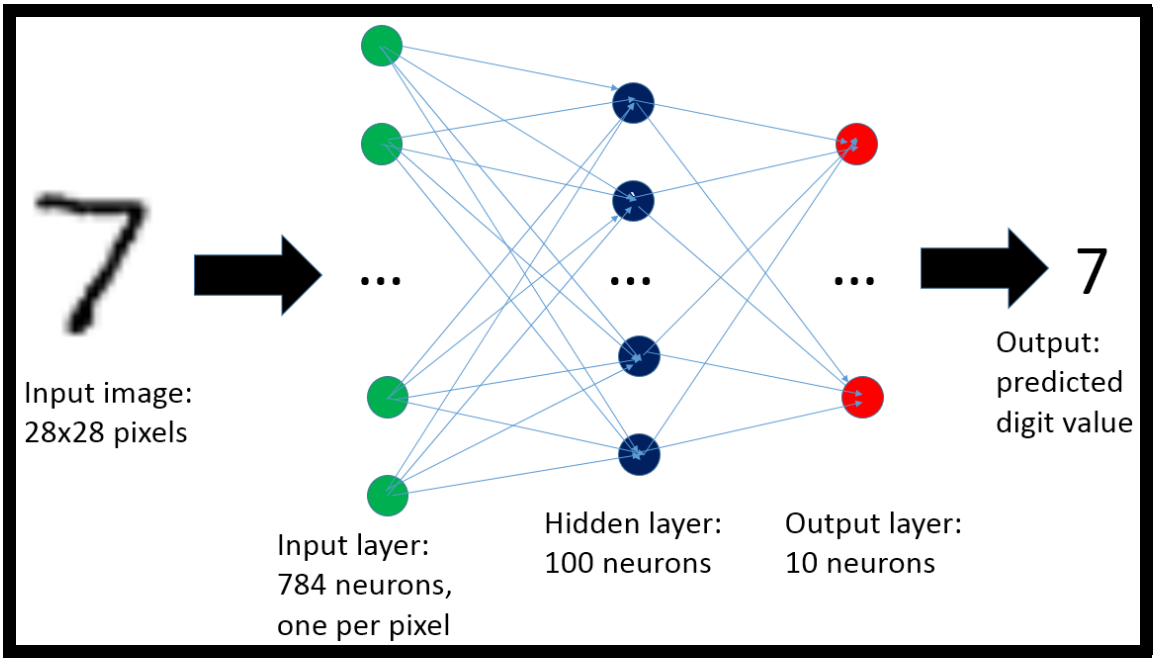

Figura 4: Diagrama de la red implementada

Es importante notar que si bien la cantidad de neuronas en la capa interna y externa está 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 cómputo 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. 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 cómputo 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 vectorial.

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

4.4.3. update_weight

Esta función actualiza los valores de un vector restándole un porcentaje de los valores de otro vector. El porcentaje descontado está dado por la variable c. Esto se utiliza en la red neuronal para ir actualizando los parámetros de las neuronas hasta que converjan a una configuración que clasifica lo mejor posible.

```
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   for(uint i = 0; i < n; i++){
3     for(uint j = 0; j < m; j++){
4       output[i * m + j] = matrix1[i * m + j] * matrix2[i * m + j];
5     }
6   }
7 }
```

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    for(uint i = 0; i < n; i++) {
3      for(uint j = 0; j < l; j++){
4        output[i * l + j] = 0;
5        for(uint k = 0; k < m; k++){
6          output[i * l + j] += matrix1[i * m + k] * matrix2[k * l + j];
7        }
8      }
9    }
10 }
```

4.5. Tests

Tanto la versión del programa que utiliza Floats como la que utiliza Doubles tienen un conjunto de tests en las funciones que implementamos en ASSEMBLER y C, esto fue para corroborar el buen funcionamiento de lo desarrollado 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 de 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 múltiplo 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 pasan dos matrices de tamaño 10×32. En este caso la dimensión que nos importa que sea múltiplo de 4 es la segunda. A parte de realizar los cálculos con SIMD aprovechamos que para el caso particular en que se usa esta matriz siempre la cantidad de filas es de 10 y realizamos una técnica de loop unrolling para mejorar los tiempos de cómputo de la versión de $ASSEMBLER$ intentando evitar el overhead del branch prediction.

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 recibe 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 Toma 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 Recibe 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 pasan dos matrices, una de dimensión 10×20 y la otra de 20×30. En este caso la dimensión que nos importa que sea múltiplo 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[5]. Los hiperparámetros usados fueron:

- cantidad de unidades de la capa oculta = 30
- mini_batch = 32
- epochs = 50

[5]Un epoch es una pasada de entrenamiento sobre un mini-batch

- 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

5.3. Análisis de los resultados

Hay varios puntos destacables en los resultados obtenidos durante la experimentación. Por un lado hay funciones que no superaron a su versión de C, como son update_weight y vector_sum, que incluso dió peor. Ambas son funciones bastante simples formadas por un solo loop, creemos que el compilador de C puede tener ciertas optimizaciones para estos casos básicos que hacen que obtenga buenos tiempos. De todas maneras algo interesante para notar en estas dos funciones es que obtuvieron tiempos ideales comparando $float$ con *double*, en estas se cumple que $float$ tarda la mitad del tiempo que *double* para la versión de $ASSEMBLER$, esto se debe probablemente a lo simple que es el código de estas funciones lo cual permite que las dos versiones sean muy similares y no se agregue gran complejidad al empezar a trabajar con 4 elementos a la vez como ocurre al implementar la versión de $float$. Esto no se dió tanto con funciones más complejas, teniendo como caso mas extremo a matrix_prod la cual dió peor con $float$ que con *double*, esto sucedió debido a que tuvimos que utilizar funciones más sofisticadas y menos eficientes para reordenar y mover los valores obtenidos cuando trabajabamos de a 4, perdiendo todo lo que habiamos ganado al paralelizar mas.

Como último comentario es importante notar como todas estas optimizaciones juntas terminan logrando una mejora significativa en la red neuronal, logrando que una epoch reduzca su tiempo de ejecución en aproximadamente un 23 %.

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 $ASSEMBLER$ con una performance superior a la que obtenían expertos en el área hace menos de una década[6], 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 las optimizaciones que realiza C fueron suficientes e incluso superiores a las mejoras que realizamos nosotros en $ASSEMBLER$ lo cual nos hizo darnos cuenta que es útil sacar el máximo provecho de las mismas antes de recaer en optimizaciones hechas a mano. De todas maneras como se pudo ver con las funciones $cost\_derivative$, $hadamard\_product$ y $matrix\_prod$, estas obtuvieron resultados considerablemente superiores a sus versiones en C, lo cual prueba que bajo ciertas circunstancias tiene sentido y es muy fructífero realizar este tipo de mejoras.

Otra conclusión importante fue que a veces paralelizar al máximo genera código más complejo lo cual termina volviendo al programa más lento y difícil de mantener por lo que puede ser incluso mejor trabajar con una concurrencia de menos operaciones a la vez.

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.

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