Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

Trabajo Práctico Final

Organización del Computador II

Noviembre 2016

Apellido y Nombre	LU	E-mail
Lovisolo, Leandro	645/11	leandro@leandro.me

1. Introducción

El objetivo de este trabajo es comparar los tiempos de entrenamiento de distintas implementaciones de una red neuronal para reconocimiento de dígitos manuscritos. Las implementaciones en cuestión son las siguientes:

- **Implementación naive**: todas las operaciones de álgebra lineal se implementan de manera naive utilizando bucles anidados que operan sobre matrices y vectores de a una coordenda por vez.
- **Implementación SIMD**: las operaciones de álgebra lineal se paralelizan a nivel instrucción utilizando operaciones SIMD siempre que sea posible.
- **Implementacón Eigen**: se utiliza la librería Eigen[1] para realizar las operaciones de álgebra lineal. Eigen es una librería C++ que implementa algoritmos de álgebra lineal de manera optimizada. Se usa frecuentemente en la industria para aplicaciones de alto rendimiento.

Todas las implementaciones fueron escritas en C++, salvo partes de la implementación SIMD que fueron escritas en assembler x86-64. Los compiladores usados fueron clang[2] y NASM[3], respectivamente.

Cada implementación es compilada con niveles de optimización O0, O1, O2 y O3. Esto arroja 12 versiones distintas de la red neuronal (3 implementaciones por 4 niveles de optimización posibles.)

Cada una de las 12 versiones resultantes se las entrena utilizando el dataset de dígitos manuscritos MNIST[4] durante exactamente la misma cantidad de épocas de entrenamiento, y se les mide el tiempo que demoran en completar dicha tarea.

Finalmente se compara el tiempo de entrenamiento de cada versión, se discuten los resultados obtenidos y se obtienen conclusiones.

2. Preliminares

La red neuronal implementada en este trabajo se basa en la red neuronal para reconocimiento de dígitos implementada en los capítulos 1[5] y 2[6] del libro Neural Networks and Deep Learning[7], de Michael Nielsen[8].

En lo siguiente se introducen el problema del reconocimiento de dígitos y los conceptos básicos de redes neuronales necesarios para este trabajo.

2.1. Reconocimiento de dígitos

Se desea implementar un sistema de reconocimiento de dígitos manuscritos, como los que se ven en la siguiente imagen.

Figura 1: Dígitos manuscritos

[1] http://eigen.tuxfamily.org/
[2] http://clang.llvm.org/
[3] http://www.nasm.us/
[4] http://yann.lecun.com/exdb/mnist/
[5] http://neuralnetworksanddeeplearning.com/chap1.html
[6] http://neuralnetworksanddeeplearning.com/chap2.html
[7] http://neuralnetworksanddeeplearning.com/
[8] http://michaelnielsen.org/

Más concretamente, dada una imagen i de 28 x 28 pixeles en escala de grises (codificados como enteros de 0 a 255), se desea estimar, para cada dígito $j \in 0, 1, \ldots, 9$, la probabilidad $p_j(i)$ que la imagen i corresponda a cada dígito j.

2.2. Datos de entrenamiento

Para resolver este problema se entrenará una red neuronal utilizando la base de datos de dígitos manuscritos MNIST[9] , que se compone de 60,000 ejemplos de entrenamiento (imágenes en escala de grises de 28 x 28 con una etiqueta que indica el dígito de 0 a 9 al que correspoden) y un conjunto de test de 10,000 imágenes etiquetadas, que no se usan para entrenar el modelo sino para medir su precisión.

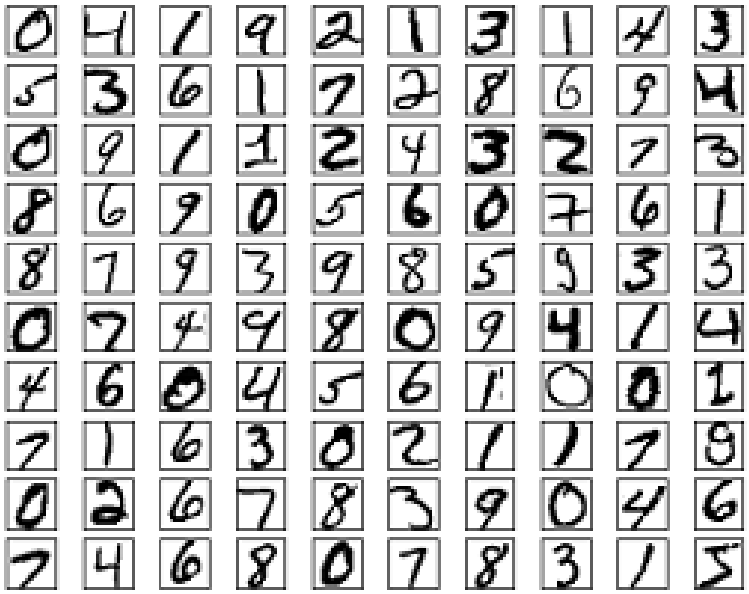

Figura 2: Dígitos de la base de datos MNIST

2.3. Introducción a redes neuronales

A continuación se hace una brevísima introducción a los conceptos básicos necesarios para entender el modelo desarrollado en este trabajo y su implementación.

2.3.1. Función sigmoide

La función sigmoide $\sigma : \mathbb{R} \to \mathbb{R}$ es una función matemática con forma de "S" (llamada curva sigmoide) definida por la siguiente fórmula:

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

Esta función permite introducir una no-linealidad en un sistema de ecuaciones, comprimiendo z en el rango $(0, 1)$ como se puede ver en la siguiente figura.

[9] yann.lecun.com/exdb/mnist/

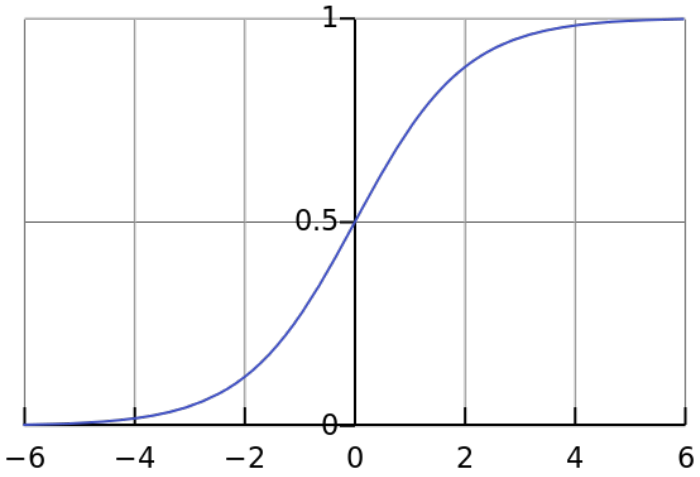

Figura 3: Curva sigmoide

La función sigmoide se utiliza para computar la salida de cada neurona en una red neuronal, como se verá a continuación.

2.3.2. Neuronas sigmoide

Una neurona sigmoide toma varias entradas $x_1, x_2, \ldots, x_n \in \mathbb{R}$ y produce una única salida.

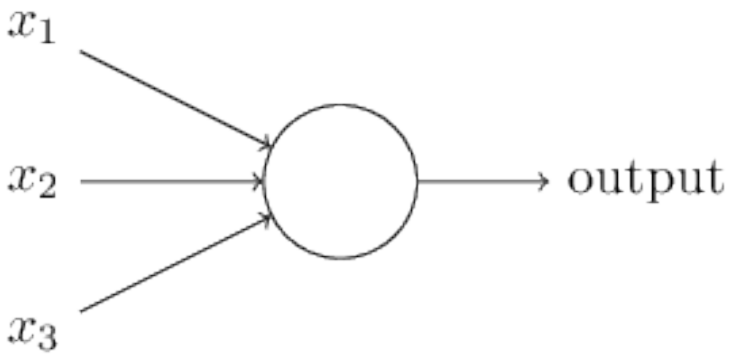

Figura 4: Neurona sigmoide

La salida *output* de la neurona sigmoide se obtiene primero computando la activación z (que se definirá a continuación), a la que se le aplica la función sigmoide σ. Es decir, $output = \sigma(z)$.

La activación z se obiene computando la suma ponderada de las entradas $x_1, \ldots, x_n$ contra un vector de pesos $w_1, w_2, \ldots, w_n \in \mathbb{R}$, a la que luego se le suma un término de sesgo $b \in \mathbb{R}$:

$$z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$

Esta combinación lineal entre $w_1, \ldots, w_n$ y $x_1, \ldots, x_n$ se puede escribir como el producto entre vectores $\boldsymbol{w} = (w_1, \ldots, w_n)$ y $\boldsymbol{x} = (x_1, \ldots, x_n)$ de la siguiente manera:

$$z = \boldsymbol{w} \cdot \boldsymbol{x} + b$$

La salida de la neurona sigmoide queda entonces definida por la siguiente fórmula:

$$output = \sigma(z) = \frac{1}{1 + exp(\sum w_i x_i + b)}$$

o equivalentemente, en notación vectorial:

$$output = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b)$$

La neurona sigmoide se puede utilizar para aproximar ciertas funciones $\mathbb{R}^n \to (0, 1)$ eligiendo $\boldsymbol{w}$ y b adecuados, como se verá a continuación.

2.3.3. Aproximando funciones con la neurona sigmoide

Existen ciertas funciones $f : \mathbb{R}^n \to (0, 1)$ que se pueden aproximar con una neurona sigmoide eligiendo adecuadamente los pesos $\boldsymbol{w} = (w_1, \ldots w_n)$ y el término de sesgo b.

Por ejemplo, para $f(x_1, \ldots, x_n) = \sigma(x_1 + 1)$, basta con elegir $\boldsymbol{w} = (1, 0, \ldots, 0)$ y $b = 1$, ya que la salida de la neurona sigmoide queda determinada de la siguiente manera:

$$\begin{aligned} output &= \sigma(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b) \\ &= \sigma(1 \cdot x_1 + 0 \cdot x_2 + \ldots + 0 \cdot x_n + 1) \\ &= \sigma(x_1 + 1) \\ &= f \end{aligned}$$

Sin embargo, esto no es siempre posible para cualquier elección de f, ya que la capacidad de aproximación de una neurona sigmoide es limitada.

A continuación veremos cómo combinar varias neuronas sigmoides para obtener una red neuronal con mayor poder de aproximación.

2.3.4. Redes neuronales

Una manera sencilla de combinar varias neuronas sigmoides es conectando la entrada $\boldsymbol{x} = (x_1, \ldots, x_n)$ a varias neuronas sigmoides en paralelo, y luego usar la salida de cada una de ellas como entradas de una neurona sigmoide adicional que las combina y produce una única salida.

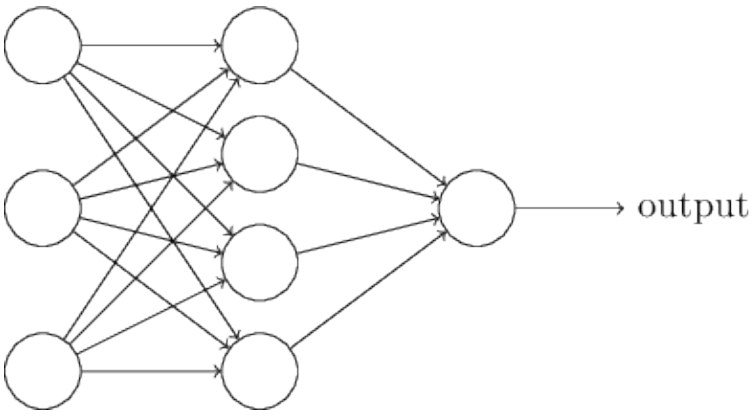

Figura 5: Red neuronal

En la red de la figura 5 se tienen 3 entradas $\boldsymbol{x} = (x_1, x_2, x_3)$ y 4 neuronas internas con vectores de pesos $\boldsymbol{w^i} = (w_1^i, w_2^i, w_3^i)$ y términos de sesgo b^i, con $i = 1, \ldots, 4$. La salida de cada neurona interna tiene la siguiente fórmula:

$$output^i = \sigma(\boldsymbol{w^i} \cdot \boldsymbol{x} + b^i)$$

Notamos $\boldsymbol{o}$ al vector de salidas de la capa de neuronas internas:

$$\boldsymbol{o} = \begin{bmatrix} outputs^1 \\ outputs^2 \\ outputs^3 \\ outputs^4 \end{bmatrix}$$

La salida de la neurona final, con vector de pesos $\boldsymbol{w}$ y sesgo b, queda entonces definida por la siguiente fórmula:

$$output = \sigma(\boldsymbol{w} \cdot \boldsymbol{o} + b)$$

A continuación se verá una manera de simplificar la notación y las cuentas necesarias para computar la salida de una red neuronal.

2.3.5. Redes neuronales expresadas como producto y suma de matrices

En la sección anterior se computó la salida $output^i$ de cada neurona sigmoide por separado. A continuación veremos cómo obtener dichos valores en un sólo paso.

Entiéndase por $\sigma(\boldsymbol{v})$ la aplicación de la función sigmoide a cada una de las coordenadas del vector $\boldsymbol{v}$. Es decir, si $\boldsymbol{v} = (v_1, \ldots, v_n)$, entonces $\sigma(\boldsymbol{v}) = (\sigma(v_1), \ldots, \sigma(v_n))$.

Sea $\boldsymbol{W}$ la matriz de pesos de la capa interna, definida como:

$$\boldsymbol{W} = \begin{bmatrix} w_1^1 & w_2^1 & w_3^1 \\ w_1^2 & w_2^2 & w_3^2 \\ w_1^3 & w_2^3 & w_3^3 \\ w_1^4 & w_2^4 & w_3^4 \end{bmatrix}$$

Sea $\boldsymbol{b}$ el vector de sesgos de la capa interna:

$$\boldsymbol{b} = \begin{bmatrix} b^1 \\ b^2 \\ b^3 \\ b^4 \end{bmatrix}$$

Utilizando $\boldsymbol{W}$ y $\boldsymbol{b}$ podemos escribir el vector $\boldsymbol{o}$ con las salidas $output^i$ de cada neurona interna de la siguiente manera:

$$\boldsymbol{o} = \sigma(\boldsymbol{W} \cdot \boldsymbol{x} + \boldsymbol{b})$$

Para corroborar esto, basta con expandir esta ecuación:

$$
\begin{aligned}
\boldsymbol{o} = \sigma(\boldsymbol{W} \cdot \boldsymbol{x} + \boldsymbol{b}) &= \sigma\left(\begin{bmatrix} w_1^1 & w_2^1 & w_3^1 \\ w_1^2 & w_2^2 & w_3^2 \\ w_1^3 & w_2^3 & w_3^3 \\ w_1^4 & w_2^4 & w_3^4 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b^1 \\ b^2 \\ b^3 \\ b^4 \end{bmatrix}\right) \\
&= \sigma\left(\begin{bmatrix} w_1^1 x_1 + w_2^1 x_2 + w_3^1 x_3 + b^1 \\ w_1^2 x_1 + w_2^2 x_2 + w_3^2 x_3 + b^2 \\ w_1^3 x_1 + w_2^3 x_2 + w_3^3 x_3 + b^3 \\ w_1^4 x_1 + w_2^4 x_2 + w_3^4 x_3 + b^4 \end{bmatrix}\right) \\
&= \begin{bmatrix} \sigma(w_1^1 x_1 + w_2^1 x_2 + w_3^1 x_3 + b^1) \\ \sigma(w_1^2 x_1 + w_2^2 x_2 + w_3^2 x_3 + b^2) \\ \sigma(w_1^3 x_1 + w_2^3 x_2 + w_3^3 x_3 + b^3) \\ \sigma(w_1^4 x_1 + w_2^4 x_2 + w_3^4 x_3 + b^4) \end{bmatrix} = \begin{bmatrix} outputs^1 \\ outputs^2 \\ outputs^3 \\ outputs^4 \end{bmatrix}
\end{aligned}
$$

El expresar la salida de una red neuronal como productos y sumas de matrices no sólo simplifica la notación sino que además permite optimizar los cómputos al aprovechar librerías de álgebra lineal que implementen eficientemente dichas operaciones.

2.3.6. Redes neuronales multicapa

La idea de combinar varias neuronas se puede generalizar a múltiples capas, como se ilustra a continuación.

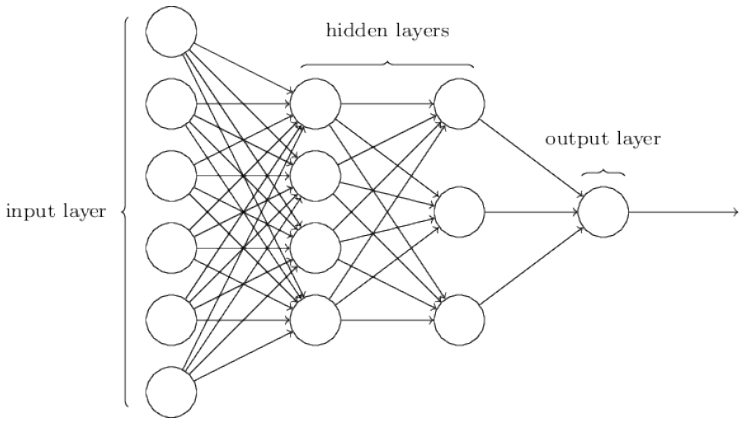

Figura 6: Red neuronal multicapa

En un modelo como en el de la figura 6 se tienen L capas de neuronas sigmoides, cada capa con su matriz de pesos $\boldsymbol{W^l}$, su vector de sesgos $\boldsymbol{b^l}$ y su vector de salidas $\boldsymbol{o^l}$ (con $1 < l < L$).

La salida de la primera capa se computar de la forma antes vista:

$$\boldsymbol{o^1} = \sigma(\boldsymbol{W^1} \cdot \boldsymbol{x} + \boldsymbol{b^1})$$

La salida de las capas subsecuentes ($l > 1$) se computan utilizando la salida de la capa anterior:

$$\boldsymbol{o^l} = \sigma(\boldsymbol{W^l} \cdot \boldsymbol{o^{l-1}} + \boldsymbol{b^l})$$

A continuación se presentará el mecanismo utilizado para ajustar los parámetros $\boldsymbol{W^l}$ y $\boldsymbol{b^l}$ de manera de obtener una red neuronal que aproxime una función de interés dado un conjunto de ejemplos de entrenamiento.

2.3.7. Algoritmo de entrenamiento

Se utiliza el algoritmo del gradiente descendente[10] para entrenar este tipo de modelos.

Sea y una función que se desea aproximar con una red neuronal con vector de pesos $\boldsymbol{w}$ y vector de sesgos $\boldsymbol{b}$. Bajo esta notación, $\boldsymbol{w}$ y $\boldsymbol{b}$ contienen los pesos y términos de sesgo de las neuronas de todas las capas de la red neuronal, respectivamente. Notaremos x a cada ejemplo de entrenamiento y $output(x)$ a la salida de la red neuronal cuando el vector $\boldsymbol{x}$ es usado como entrada.

Sea $C(\boldsymbol{w}, \boldsymbol{b})$ una función de costo que da una medida del error que se comete al aproximar y con una red neuronal. Una función de costo tal es la siguiente, conocida como error cuadrático medio:

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

Es decir, C acumula el error cuadrático que se comete al aproximar y con la red neuronal de parámetros $\boldsymbol{w}$ y $\boldsymbol{b}$ para cada ejemplo de entrenamiento $\boldsymbol{x}$.

El objetivo del algoritmo del gradiente descendente es hallar $\boldsymbol{w}$ y $\boldsymbol{b}$ tales que $C(\boldsymbol{w}, \boldsymbol{b})$ sea lo más pequeño posible.

La idea del algoritmo es computar el vector gradiente[11] $\nabla C(\boldsymbol{w}, \boldsymbol{b})$ de la función C en el punto $(\boldsymbol{w}, \boldsymbol{b})$, que indica la dirección de mayor crecimiento de C en el punto $(\boldsymbol{w}, \boldsymbol{b})$. Una vez obtenido $\nabla C(\boldsymbol{w}, \boldsymbol{b})$, se mueve $(\boldsymbol{w}, \boldsymbol{b})$ en esa dirección por una pequeña cantidad, bajo la hipótesis que $C(\boldsymbol{w}, \boldsymbol{b}) \leq C((\boldsymbol{w}, \boldsymbol{b}) - \gamma \nabla C(\boldsymbol{w}, \boldsymbol{b}))$ para algún γ suficientemente pequeño. Esto se repite hasta alcanzar algún criterio de parada, usualmente un número máximo de iteraciones o una disminución del error entre paso y paso que caiga por debajo de cierto umbral.

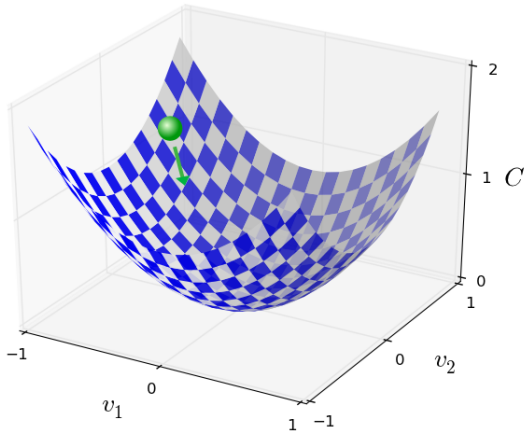

Figura 7: Gradiente descendente

Notar que este algoritmo requiere que la función de costo C sea diferenciable.

En la práctica se usa una variante de este algoritmo, denominada gradiente descendente estocástico[12], que en cada iteración computa el error del modelo contra varios minibaches del conjunto de ejemplos de entrenamiento y luego los promedia, en lugar de computar el error contra todos los ejemplos de entrenamiento en un sólo paso. Esto mejora notablemente el tiempo que se demora en entrenar un modelo.

[10] https://en.wikipedia.org/wiki/Gradient_descent
[11] https://en.wikipedia.org/wiki/Gradient
[12] https://en.wikipedia.org/wiki/Stochastic_gradient_descent

Para computar ∇C se utiliza el algoritmo de propagación hacia atrás[13] . Una buena explicación del mismo puede hallarse en el libro Neural Networks and Deep Learning, capítulo 2[14] .

3. Red neuronal para reconocimiento de dígitos manuscritos

La red neuronal implementada en este trabajo recibe en la entrada una imagen en escala de grises de 28×28 pixeles, donde cada pixel es representado por un valor entre 0 y 255. Produce a la salida un vector $\boldsymbol{v}$ de 10 coordenadas tal que para cada dígito $j \in 0, 1, \ldots, 9$, la coordenada v_j indica la probabilidad estimada p_j que la imagen recibida a la entrada corresponda al dígito j.

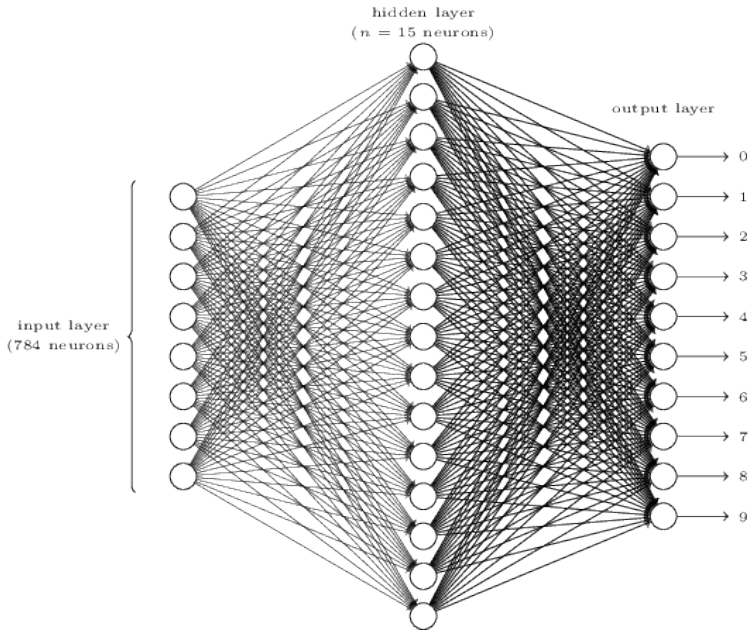

Figura 8: Arquitectura para reconocimiento de dígitos manuscritos

La arquitectura de la misma, ilustrada en la figura 8, se compone de una capa de entrada de 784 neuronas, una para cada uno de los 28×28 pixeles de entrada, una capa interna de 30 neuronas, y una capa de salida de 10 neuronas, una para cada probabilidad p_j.

4. Arquitectura

La arquitectura general del código escrito en este trabajo se compone de la siguiente jerarquía de clases C++:

- `BaseMatrix`: clase abstracta que representa matrices en $\mathbb{R}^{m \times n}$. Declara métodos virtuales para realizar operaciones sobre matrices (producto entre matrices, producto escalar, suma, etc.) Requiere que las coordenadas de la matriz sean números de punto flotante de 32 bits.
  - `NaiveMatrix`: Implementa los métodos virtuales de BaseMatrix de forma naive, utilizando bucles anidados que operan sobre una coordenada por vez.
  - `SimdMatrix`: Delega la implementaciónde los métodos virtuales de BaseMatrix en rutinas escritas en assembler que utilizan instrucciones SIMD para realizar las mismas operaciones de a 4 coordenadas por vez cuando sea posible.

[13] https://en.wikipedia.org/wiki/Backpropagation
[14] http://neuralnetworksanddeeplearning.com/chap2.html

    - `EigenMatrix`: Delega la implementación de los métodos virtuales de BaseMatrix en la librería Eigen. Más concretamente, mantiene una instancia de la clase Eigen::MatrixXf en un campo privado y simplemente delega cada operación al método equivalente en dicha clase.

- `Network<Matrix>`: Implementa la red neuronal de reconocimiento de dígitos manuscritos. Esta clase es un template C++ que recibe un parámetro de template `Matrix`, que debe corresponder a alguna de las subclases de BaseMatrix. Las tres implementaciones posibles de la red neuronal se obtienen instanciando `Network<NaiveMatrix>`, `Network<SimdMatrix>` y `Network<EigenMatrix>`.

Todas estas clases están definidas en los archivos `matrix.h` y `network.h`.

Además, se declaran algunas funciones en `simd_matrix.h` que son implementadas en `simd_matrix.asm` siguiendo la convención de llamadas del lenguaje C.

5. Implementación

A continuación se explican los detalles implementativos de `BaseMatrix` y sus subclases.

5.1. BaseMatrix

La clase `BaseMatrix` establece una interfaz para operaciones de álgebra lineal entre matrices de números de punto flotante de precisión simple (tipo `float` en C++.) En ella se definen métodos de utilidad compartidos por todas las subclases, como inicializadores, acceso a coeficientes, etc. Además, declara métodos virtuales que todas las subclases deberán implementar. Los principales se listan a continuación:

- Adición:
  `void BaseMatrix::operator+=(const Matrix& other)`
- Substracción:
  `void BaseMatrix::operator-=(const Matrix& other)`
- Producto coordenada a coordenada:
  `Matrix BaseMatrix::CoeffWiseProduct(const Matrix& other)`
- Producto escalar:
  `void BaseMatrix::operator*=(float c)`
- Producto entre matrices:
  `void BaseMatrix::operator*=(const Matrix& other)`
- Transposición:
  `Matrix BaseMatrix::Transpose()`

Nota: el tipo `Matrix` es un template parameter que se reemplaza por el nombre de cada subclase que hereda de `BaseMatrix`. Por ejemplo, en la clase `SimdMatrix` la signatura del método para adición pasa a ser `SimdMatrix SimdMatrix::CoeffWiseProduct(const SimdMatrix &other)`.

A continuación se ilustra la definición del resto de la clase `BaseMatrix`, que será útil como referencia al introducir el código para las subclases en las siguientes secciones.

```
template <class Matrix>
class BaseMatrix {
 public:
  BaseMatrix(uint rows, uint cols) : rows_(rows), cols_(cols) {}

  inline uint rows() const { return rows_; }
  inline uint cols() const { return cols_; }
```

```
  // Acceso a coeficientes
  virtual float& operator()(uint i, uint j) = 0;
  virtual float operator()(uint i, uint j) const = 0;

  // Operaciones de álgebra lineal (omitidas por brevedad)
  ...

 protected:
  uint rows_;
  uint cols_;
};
```

5.2. NaiveMatrix

La clase `NaiveMatrix` mantiene los coeficientes de la matriz en un vector `m_` de numeros de punto flotante (tipo `vector<float>`), como se observa a continuación:

```
class NaiveMatrix : public BaseMatrix<NaiveMatrix> {
  ...

 private:
  std::vector<float> m_;  // los coeficientes de la matriz se mantienen aquí
};
```

Los coeficientes son ordenados de izquierda a derecha y de arriba a abajo. Es decir, el valor para la coordenada (i, j) se halla en `m_[i * columns + j]`.

Esta clase implementa todas las operaciones de forma sencilla y sin ningún tipo de optimización, usando bucles anidados que operan sobre las coordenadas de la matriz de a una por vez.

A continuación se lista el código de las mismas.

5.2.1. Adición

Primero se verifica que ambas matrices tengan el mismo tamaño, luego se suman los coeficientes uno por uno.

```
void NaiveMatrix::operator+=(const NaiveMatrix& other) {
  assert(rows_ == other.rows_);
  assert(cols_ == other.cols_);
  for(uint i = 0; i < rows_ * cols_; i++) {
    m_[i] += other.m_[i];
  }
}
```

5.2.2. Substracción

Análogo a la operación de adición.

```
void NaiveMatrix::operator-=(const NaiveMatrix& other) {
  assert(rows_ == other.rows_);
  assert(cols_ == other.cols_);
  for(uint i = 0; i < rows_ * cols_; i++) {
    m_[i] -= other.m_[i];
```

```
    }
  }
```

5.2.3. Producto coordenada a coordenada

Se verifica que ambas matrices tengan el mismo tamaño y luego se multiplican los coeficientes uno a uno.

```
NaiveMatrix NaiveMatrix::CoeffWiseProduct(const NaiveMatrix& other) const {
  assert(rows_ == other.rows_);
  assert(cols_ == other.cols_);
  NaiveMatrix res(*this);
  for(uint i = 0; i < res.m_.size(); i++) {
    res.m_[i] *= other.m_[i];
  }
  return res;
}
```

5.2.4. Producto escalar

Se multiplican los coeficientes por una constante, de a uno por vez.

```
void NaiveMatrix::operator*=(float c) {
  for(uint i = 0; i < m_.size(); i++) {
    m_[i] *= c;
  }
}
```

5.2.5. Producto entre matrices

Se verifica que los tamaños de ambas matrices sean compatibles, luego se crea la matriz destino, y para cada coordenada (i, j) en la matriz destino, se computa el producto interno entre la fija i de la primera matriz y la columna j en la segunda matriz, de a una coordenada por vez.

```
void NaiveMatrix::operator*=(const NaiveMatrix& other) {
  assert(cols_ == other.rows_);
  uint new_rows = rows_;
  uint new_cols = other.cols_;
  vector<float> new_m(new_rows * new_cols, 0.);
  NaiveMatrix transpose = other.Transpose();
  for(uint i = 0; i < new_rows; i++) {
    for(uint j = 0; j < new_cols; j++) {
      for(uint k = 0; k < cols_; k++) {
        new_m[new_cols * i + j] += operator()(i, k) * transpose(j, k);
      }
    }
  }
  m_ = new_m;
  rows_ = new_rows;
  cols_ = new_cols;
}
```

5.2.6. Transposición

Se crea la matriz destino y se copia cada coeficiente (i, j) en las posición (j, i) en la matriz destino, de a uno por vez.

```
NaiveMatrix NaiveMatrix::Transpose() const {
  NaiveMatrix res(cols_, rows_);
  for(uint i = 0; i < rows_; i++) {
    for(uint j = 0; j < cols_; j++) {
      res(j, i) = operator()(i, j);
    }
  }
  return res;
}
```

5.3. SimdMatrix

La clase `SimdMatrix` delega las operaciones sobre matrices a rutinas escritas en assembler basadas en los algoritmos de `NaiveMatrix`, pero que hacen uso de los registros XMM del procesador operando de a cuatro coordenadas por vez.

Para los casos donde las matrices tienen un número de filas o columnas inferior a 4 (que dificulta el uso de registros XMM) se delega a una implementación ad-hoc en C++ que realiza las operaciones de forma idéntica a `NaiveMatrix`.

Al igual que `NaiveMatrix`, mantiene los coeficientes en un `vector<float>`. Las rutinas escritas en assembler reciben un puntero al bloque de memoria administrado por la instancia de `vector<float>` y operan sobre este transparentemente.

```
class SimdMatrix : public BaseMatrix<SimdMatrix> {
  ...

 private:
  std::vector<float> m_;  // los coeficientes de la matriz se mantienen aquí
};
```

A continuación se muestra el código de las rutinas escritas en assembler.

5.3.1. Adición

Esta rutina implementa una función C con la siguiente signatura:

```
void simd_matrix_addition(uint size,
                          float *m,
                          const float *n)
```

Dadas dos matrices M y N en $\mathbb{R}^{k \times q}$ con $k \times q \geq 4$, esta rutina recibe un parámetro `size` $= k \times q$ y dos punteros `m` y `n` a arreglos de floats de tamaño `size` con los coeficientes de M y N, respectivamente, computa $M' = M + N$ y escribe en `m` los coeficientes de M'.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; void simd_matrix_addition(uint size,       // rdi ;;
;;                           float* m,        // rsi ;;
;;                           const float* n)  // rdx ;;
;; Assumes size >= 4.                                ;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

simd_matrix_addition:
  push rbp
  mov rbp, rsp
  push rbx
  push r12
  push r13
  push r14
  push r15

  ; Clear high 32 bits of rdi
  mov edi, edi

  ; Store in r12 the maximum offset we want use with the SIMD instruction
  mov rax, rdi              ; rax = size
  sub rax, 4                ; rax = size - 4
  mov rbx, 4                ; rbx = 4
  push rdx
  mul rbx                   ; rax = 4 * (size - 4)
  pop rdx
  mov r12, rax              ; r12 = 4 * (size - 4)

  ; Initialize r13 as the offset register used in main loop
  xor r13, r13              ; r13 = 0

  ; Main loop
simd_matrix_addition_loop:
  movups xmm0, [rsi + r13]  ; Copy values at current offset in matrix m
  movups xmm1, [rdx + r13]  ; Copy values at current offset in matrix n
  addps xmm0, xmm1          ; Execute SIMD op
  movups [rsi + r13], xmm0  ; Store results at current offset in matrix m
  movups xmm2, xmm0         ; Backup xmm0 in xmm2 (see case 3 below)
  mov r14, r13              ; Backup offset register r13 in r14

  ; Compute difference between current and maximum offset
  mov r15, r12              ; r15 = max offset
  sub r15, r13              ; r15 = max offset - current offset

  ; Case 1: current offset = max offset
  jz exit_simd_matrix_addition_loop

  ; Case 2: 4 or more elements left
  cmp r15, 16
  jge simd_matrix_addition_loop_next

  ; Case 3: 1, 2 or 3 elements left
  movups xmm0, [rsi + r12]  ; Copy values at max offset in matrix m
  movups xmm1, [rdx + r12]  ; Copy values at max offset in matrix n
  addps xmm0, xmm1          ; Execute SIMD op
  movups [rsi + r12], xmm0  ; Store results at max offset in matrix m
  movups [rsi + r14], xmm2  ; Restore backed-up values
  jmp exit_simd_matrix_addition_loop

simd_matrix_addition_loop_next:
  add r13, 16               ; Advance 4 elements
  jmp simd_matrix_addition_loop
```

```
exit_simd_matrix_addition_loop:
  pop r15
  pop r14
  pop r13
  pop r12
  pop rbx
  pop rbp
  ret
```

Esencialmente el algoritmo se compone de un loop `simd_matrix_addition_loop` que lee de a cuatro elementos de M y N por vez, que son almacenados en `xmm0` y `xmm1` respectivamente, luego se suman uno a uno con `addps` y el resultado se guarda en M pisando los 4 valores actuales.

En el caso donde restan 1, 2 o 3 valores, se avanza esa cantidad de posiciones en lugar de 4, se cargan los últimos 4 coeficientes de ambas matrices en `xmm0` y `xmm1` y se computa la suma con `addps`. Esto recomputa y pisa algunos de los valores obtenidos en la iteración anterior, pero no representa un problema ya que se trata de operaciones coeficiente a coeficiente que no dependen de otros coeficientes en la matriz original.

Nota: el código assembler mostrado aquí es en realidad generado por un macro NASM (ver macro `coeff_wise_vector_to_vector_op` definido en `simd_matrix.asm`) que toma una instrucción XMM como parámetro (por ejemplo `addps`), y genera un procedimiento para operaciones coeficiente a coeficiente usando dicha instrucción. Esto es así pues la diferencia entre las implementaciones de las operaciones de adición, substracción y producto coordenada a coordenada son idénticas salvo por una única instrucción XMM que define la operación (`addps`, `subps` y `mulps`, respectivamente.)

5.3.2. Substracción

Esta rutina implementa una función C con la siguiente signatura:

```
void simd_matrix_subtraction(uint size,
                             float *m,
                             const float *n)
```

Dadas dos matrices M y N en $\mathbb{R}^{k \times q}$ con $k \times q \geq 4$, esta rutina recibe un parámetro `size` $= k \times q$ y dos punteros `m` y `n` a arreglos de floats de tamaño `size` con los coeficientes de M y N, respectivamente, computa $M' = M - N$ y escribe en `m` los coeficientes de M'.

El código assembler es idéntico a la operación de adición, salvo que se reemplaza la instrucción `addps` por `subps`.

5.3.3. Producto coordenada a coordenada

Esta rutina implementa una función C con la siguiente signatura:

```
void simd_matrix_coeff_wise_product(uint size,
                                    float *m,
                                    const float *n)
```

Dadas dos matrices M y N en $\mathbb{R}^{k \times q}$ con $k \times q \geq 4$, esta rutina recibe un parámetro `size` $= k \times q$ y dos punteros `m` y `n` a arreglos de floats de tamaño `size` con los coeficientes de M y N, respectivamente, computa $M'_{i,j} = M_{i,j} \times N_{i,j}$ para cada (i, j) y escribe en `m` los coeficientes de M'.

El código assembler es idéntico a la operación de adición, salvo que se reemplaza la instrucción `addps` por `mulps`.

5.3.4. Producto escalar

Esta rutina implementa una función C con la siguiente signatura:

```
void simd_matrix_scalar_product(uint size,
                                float* m,
                                float c)
```

Dadas una constante $c \in \mathbb{R}$ y una matriz M en $\mathbb{R}^{k \times q}$ con $k \times q \geq 4$, esta rutina recibe un parámetro `size` $= k \times q$ y un puntero `m` a un arreglo de floats de tamaño `size` con los coeficientes de M, computa $M' = c \times M$ y escribe en `m` los coeficientes de M'.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; void simd_matrix_scalar_product(uint size,  // rdi  ;;
;;                                  float* m,   // rsi  ;;
;;                                  float c)    // xmm0 ;;
;; Assumes size >= 4.                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

simd_matrix_scalar_product:
  push rbp
  mov rbp, rsp
  push rbx
  push r12
  push r13
  push r14
  push r15

  ; Clear high 32 bits of rdi
  mov edi, edi

  ; Replicate value of xmm0
  pshufd xmm0, xmm0, 0      ; xmm0 = c | c | c | c

  ; Store in r12 the maximum offset we want use with mulps instruction
  mov rax, rdi              ; rax = size
  sub rax, 4                ; rax = size - 4
  mov rbx, 4                ; rbx = 4
  push rdx
  mul rbx                   ; rax = 4 * (size - 4)
  pop rdx
  mov r12, rax              ; r12 = 4 * (size - 4)

  ; Initialize r13 as the offset register used in scalar product loop
  xor r13, r13              ; r13 = 0

  ; Main loop
scalar_product_loop:
  movups xmm1, [rsi + r13]  ; Copy values at current offset in matrix m
  mulps xmm1, xmm0          ; Multiply values
  movups [rsi + r13], xmm1  ; Store results at current offset in matrix m
  movups xmm2, xmm1         ; Backup xmm0 in xmm2 (see case 3 below)
  mov r14, r13              ; Backup offset register r13 in r14

  ; Compute difference between current and maximum offset
  mov r15, r12              ; r15 = max offset
  sub r15, r13              ; r15 = max offset - current offset
```

```
  ; Case 1: current offset = max offset
  jz exit_scalar_product_loop

  ; Case 2: 4 or more elements left
  cmp r15, 16
  jge scalar_product_loop_next

  ; Case 3: 1, 2 or 3 elements left
  movups xmm1, [rsi + r12]  ; Copy values at max offset in matrix m
  mulps xmm1, xmm0          ; Add up values
  movups [rsi + r12], xmm1  ; Store results at max offset in matrix m
  movups [rsi + r14], xmm2  ; Restore backed-up values
  jmp exit_scalar_product_loop

scalar_product_loop_next:
  add r13, 16               ; Advance 4 elements
  jmp scalar_product_loop

exit_scalar_product_loop:
  pop r15
  pop r14
  pop r13
  pop r12
  pop rbx
  pop rbp
  ret
```

El algoritmo replica la constante c en los 4 floats contenidos en `xmm0` y entra en el bucle `scalar_product_loop`, donde se leen en `xmm1` los coeficientes de M de a cuatro por vez, se los multiplica por c ejecutando `mulps xmm1, xmm0` y se almacena el resultado en M pisando los 4 valores anteriores.

Al final de cada iteración se guarda en `xmm2` una copia de los resultados obtenidos, y se guarda en `r14` el offset actual. Esta copia se usa para restaurar posibles valores pisados durante la última iteración del bucle.

En el caso donde restan 1, 2 o 3 valores, se avanza esa cantidad de posiciones en lugar de 4, se cargan los últimos 4 coeficientes de M en `xmm1` y se los multiplica por c con `mulps`. Esto vuelve a multiplicar por c algunos de los coeficientes tratados en la iteración anterior. Por ese motivo se los restaura a su estado anterior volcando el contenido del registro `xmm2` en el offset indicado en `r14`.

5.3.5. Producto entre matrices

Esta rutina implementa una función C con la siguiente signatura:

```
void simd_matrix_product(uint rows1,
                         uint cols1,
                         uint cols2,
                         const float* m,
                         const float* nt,
                         float* p)
```

Dadas dos matrices $M \in \mathbb{R}^{k \times q}$ y $N \in \mathbb{R}^{q \times r}$ (con $k, q, r \geq 4$), esta rutina recibe los siguientes parámetros:

- `rows1` $= k$,
- `cols1` $= q$,

- `cols2` $= r$,
- un puntero `m` a un arreglo de floats de tamaño $k \times q$ con los coeficientes de M,
- un puntero `nt` a un arreglo de floats de tamaño $r \times q$ con los coeficientes de N^t (la matriz transpuesta de N),
- un puntero `p` a un arreglo de floats de tamaño $k \times r$.

La rutina computa el producto matricial $P = M \times N$ y escribe en `p` los coeficientes de P.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; void simd_matrix_product(uint rows1,      // rdi ;;
;;                          uint cols1,      // rsi ;;
;;                          uint cols2,      // rdx ;;
;;                          const float* m,  // rcx ;;
;;                          const float* nt, // r8  ;;
;;                          float* p)        // r9  ;;
;;                                                  ;;
;; Computes the product between matrices m and n,   ;;
;; and stores the results in p.                     ;;
;;                                                  ;;
;; Assumes:                                         ;;
;;  - rows1 >= 4, cols1 >= 4 and cols2 >= 4.        ;;
;;  - rows2 = cols1 (note that rows2 is not         ;;
;;    provided as an argument).                     ;;
;;  - matrix nt is the transpose of n.              ;;
;;  - cols2 corresponds to the number of columns in ;;
;;    the original matrix n *before* transposition. ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

simd_matrix_product:
  push rbp
  mov rbp, rsp
  push rbx
  push r12
  push r13
  push r14
  push r15

  ; Clear high 32 bits of rdi, rsi and rdx
  mov edi, edi
  mov esi, esi
  mov edx, edx

  ; Row index (i) in the resulting matrix
  xor r12, r12

  ; Iterate over rows in the resulting matrix
product_rows_loop:

  ; Column index (j) in the resulting matrix
  xor r13, r13

  ; Iterate over columns in the resulting matrix
product_cols_loop:

  ; Compute value for cell (i,j) in the result matrix and store it in xmm2
  call product_compute_ij
```

```
  ; Save value in xmm2 in cell (i,j) in the result matrix
  call product_save_ij

  ; Increase col index and iterate
  inc r13
  cmp r13, rdx
  jne product_cols_loop

  ; Increase row index and iterate
  inc r12
  cmp r12, rdi
  jne product_rows_loop

  pop r15
  pop r14
  pop r13
  pop r12
  pop rbx
  pop rbp
  ret
```

La rutina itera sobre las filas de la matriz resultante P (bucle `product_rows_loop`) y sobre las columnas de la misma (bucle `product_cols_loop`). Para cada coordenada (i, j) en P, invoca la rutina `product_compute_ij` que computa el valor $P_{i,j}$ y lo guarda en `xmm2`, y luego invoca la rutina `product_save_ij` que guarda el valor en `xmm2` en la posición correspondiente a $P_{i,j}$ en `p`.

Ambas rutinas se elaboran a continuación.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Procedure product_compute_ij computes the value for cell (i,j) in      ;;
;; the resulting matrix. Expects rdi = rows, rsi=cols1, rcx=pointer to m, ;;
;; r8=pointer to nt (transpose of n), r9=pointer to resulting matrix,     ;;
;; r12=i, r13=j.                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

product_compute_ij:

  ; Initialize accumulator
  pxor xmm2, xmm2

  ; Internal loop index (k)
  xor r14, r14

product_internal_loop:

  ; Load m[i, k..k+3] into xmm0 and n[k..k+3, j] into xmm1
  call product_ijk_fetch

  ; Compute dot-product between xmm0 and xmm1 and store it in xmm0
  call product_ijk_reduce

  ; Accumulate results
  addss xmm2, xmm0

  ; Compute number of remaining cells left
  mov r15, rsi              ; r14 = cols1
  sub r15, r14              ; r14 = cols1 - current cell
```

```
  sub r15, 4               ; r14 = cols1 - current cell - 4

  ; Case 1: no cells left
  jz exit_product_internal_loop

  ; Case 2: 4 or more cells left
  cmp r15, 4
  jge product_internal_loop_next

  ; Case 3: 1, 2 or 3 columns left
  mov r14, rsi             ; k = r14 = cols1
  sub r14, 4               ; k = r14 = cols1 - 4

  call product_ijk_fetch ; Load m[i, cols1-4..cols1] into xmm0
                         ; and  n[cols1-4..cols1, j] into xmm1

  ; 1 column left
  cmp r15, 1
  je product_internal_loop_1_column_left

  ; 2 columns left
  cmp r15, 2
  je product_internal_loop_2_columns_left

  ; 3 columns left
  jmp product_internal_loop_3_columns_left

  ; Case 3: 1 column left
product_internal_loop_1_column_left:

  ; Compute the product between the last float in
  ; xmm0 and xmm1 and store it in xmm0
  call product_ijk_reduce_1_column_left

  ; Finish reduction
  jmp product_internal_loop_special_case_finish_reduction

  ; Case 3: 2 columns left
product_internal_loop_2_columns_left:

  ; Compute dot-product between the last 2 floats in
  ; xmm0 and xmm1 and store it in xmm0
  call product_ijk_reduce_2_columns_left

  ; Finish reduction
  jmp product_internal_loop_special_case_finish_reduction

  ; Case 3: 3 columns left
product_internal_loop_3_columns_left:

  ; Compute dot-product between the last 3 floats in
  ; xmm0 and xmm1 and store it in xmm0
  call product_ijk_reduce_3_columns_left

product_internal_loop_special_case_finish_reduction:

  ; Accumulate results
  addss xmm2, xmm0
```

```
  jmp exit_product_internal_loop

product_internal_loop_next:
  add r14, 4                ; Advance 4 cells
  jmp product_internal_loop

exit_product_internal_loop:
  ret
```

La rutina `product_compute_ij` computa el valor para $P_{i,j}$, que es lo mismo que computar el producto interno entre la i-ésima fila de M y la j-éstima columna de N.

Sin embargo, para mejorar la localidad de caché y aprovechar las operaciones con registros XMM, se usa la matriz transpuesta N^t en lugar de N, computándose el producto interno entre la i-éstima fila de M y la j-ésima fila de N^t.

La rutina, pues, ejecuta un bucle `product_internal_loop` que itera sobre los coeficientes de ambos vectores fila de a cuatro por vez (índice k almacenado en `r14`), invoca `product_ijk_fetch` que carga $M_{i,k...k+3}$ en `xmm0` y $N^t_{j,k...k+3}$ en `xmm1`, invoca `product_ijk_reduce` que computa el producto interno entre `xmm0` y `xmm1` y acumula este resultado parcial en `xmm2`.

En los casos donde resten 1, 2 o 3 valores en la última iteración, en lugar de `product_ijk_reduce` se invoca `product_ijk_reduce_1_column_left`, `product_ijk_reduce_2_columns_left` o `product_ijk_reduce_3_columns_left` según corresponda.

Finalmente el valor para $P_{i,j}$ se retorna en el registro `xmm2`.

A continuación se explican dichas rutinas.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Procedure product_ijk_fetch loads m[i, cols1-4..cols1] into xmm0 ;;
;; and n[cols1-4..cols1, j] into xmm1. Expects r12=i, r13=j, r14=k, ;;
;; rsi=cols1, rcx=pointer to m, r8=pointer to nt (transpose of n).  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

product_ijk_fetch:

  ; Compute offset for m(i, k)
  mov rax, r12              ; rax = i
  push rdx
  mul rsi                   ; rax = i * cols1
  add rax, r14              ; rax = (i * cols1) + k
  mov rbx, 4                ; rbx = 4
  mul rbx                   ; rax = 4 * ((i * cols1) + k)
  pop rdx

  ; Store m(i, k..k+3) in xmm0
  movups xmm0, [rcx + rax]  ; Copy values at current offset in matrix m

  ; Compute offset for n^t(j, k)
  mov rax, r13              ; rax = j
  push rdx
  mul rsi                   ; rax = j * cols1
  add rax, r14              ; rax = (j * cols1) + k
  mov rbx, 4                ; rbx = 4
  mul rbx                   ; rax = 4 * ((j * cols1) + k)
  pop rdx

  ; Store nt(j, k..k+3) in xmm1
```

```
  movups xmm1, [r8 + rax]  ; Copy values at current offset in matrix nt

  ret
```

La rutina `product_ijk_fetch` computa el desplazamiento en `m` donde se hallan los valores correspondientes a $M_{i,k\ldots k+3}$ y luego carga estos valores en `xmm0`. Luego hace lo mismo para `nt` y $N^t_{j,k\ldots k+3}$ y guarda estos valores en `xmm1`.

A continuación se explica la rutina `product_ijk_reduce` y sus variantes para 1, 2 y 3 valores restantes.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Procedure product_ijk_reduce computes the dot product between ;;
;; xmm0 and xmm1 and stores it in the low dword in xmm0.         ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

product_ijk_reduce:
  dpps xmm0, xmm1, 0xF1
  ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Procedure product_ijk_reduce computes the dot product between ;;
;; the high dword in  xmm0 and xmm1 and stores it in the low     ;;
;; dword in xmm0.                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

product_ijk_reduce_1_column_left:
  dpps xmm0, xmm1, 0x81
  ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Procedure product_ijk_reduce computes the dot product between ;;
;; the two higher dwords in  xmm0 and xmm1 and stores it in the  ;;
;; low dword in xmm0.                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

product_ijk_reduce_2_columns_left:
  dpps xmm0, xmm1, 0xC1
  ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Procedure product_ijk_reduce computes the dot product between ;;
;; the three higher dwords in  xmm0 and xmm1 and stores it in    ;;
;; the low dword in xmm0.                                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

product_ijk_reduce_3_columns_left:
  dpps xmm0, xmm1, 0xE1
  ret
```

La rutina `product_ijk_reduce` y sus variantes simplemente computan el producto interno entre los valores correspondientes en `xmm0` y `xmm1` y devuelven el resultado en `xmm0`.

El último paso de la rutina `simd_matrix_product` consiste en guardar en `p` el valor para $P_{i,j}$ obtenido con `product_compute_ij`. Esto se realiza por medio de la rutina `product_save_ij`, que se explica a continuación.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Procedure product_save_ij stores the computed value for (i,j) ;;
```

```
;; in the resulting matrix. Expects r12=i, r13=j, rdx=cols2,     ;;
;; r9=pointer to the resulting matrix.                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

product_save_ij:

  ; Compute offset for p(i, j)
  mov rax, r12              ; rax = i
  push rdx
  mul rdx                   ; rax = i * cols2
  add rax, r13              ; rax = (i * cols2) + j
  mov rbx, 4                ; rbx = 4
  mul rbx                   ; rax = 4 * ((i * cols2) + j)
  pop rdx

  ; Store xmm0 in p(i, j)
  movss [r9 + rax], xmm2    ; Copy values at current offset in matrix m

  ret
```

La rutina `product_save_ij` calcula el desplazamiento para (i, j) en `p` y guarda allí el valor para $P_{i,j}$ contenido en `xmm2`, que fue computado en `product_compute_ij`.

5.3.6. Transposición

Esta rutina implementa una función C con la siguiente signatura:

```
void simd_matrix_transpose(uint rows,
                           uint cols,
                           const float* m,
                           float* n)
```

Dada una matriz M en $\mathbb{R}^{k \times q}$ con $k \geq 4$, esta rutina recibe los siguientes parámetros:

- `rows` $= k$,
- `cols` $= q$,
- un puntero `m` a un arreglo de floats de tamaño $k \times q$ con los coeficientes de M,
- un puntero `n` a un arreglo de floats de tamaño $q \times k$.

La rutina computa la matriz transpuesta $M^t \in \mathbb{R}^{q \times k}$ y escribe en `n` los coeficientes de M^t.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; void simd_matrix_transpose(uint rows,       // rdi ;;
;;                            uint cols,       // rsi ;;
;;                            const float* m,  // rdx ;;
;;                            float* n)        // rcx ;;
;; Assumes rows >= 4.                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

simd_matrix_transpose:
  push rbp
  mov rbp, rsp
  push rbx
```

```
  push r12
  push r13
  push r14
  push r15

  ; Clear high 32 bits of rdi and rsi
  mov edi, edi
  mov esi, esi

  ; Column index (j)
  xor r13, r13              ; r13 = 0

  ; Iterate over columns
transpose_cols_loop:

  ; Row index (i)
  xor r12, r12              ; r12 = 0

  ; Iterate over rows
transpose_rows_loop:

  ; Copy m[i..i+3, j] to n[j, i..i+3]
  call transpose_ij

  ; Compute number of remaining rows left
  mov r14, rdi              ; r14 = rows
  sub r14, r12              ; r14 = rows - current row
  sub r14, 4                ; r14 = rows - current row - 4

  ; Case 1: no rows left
  jz exit_transpose_rows_loop

  ; Case 2: 4 or more rows left
  cmp r14, 4
  jge transpose_rows_loop_next

  ; Case 3: 1, 2 or 3 rows left
  mov r12, rdi              ; i = r12 = rows
  sub r12, 4                ; i = r12 = rows - 4
  call transpose_ij
  jmp exit_transpose_rows_loop

transpose_rows_loop_next:
  add r12, 4                ; Advance 4 rows
  jmp transpose_rows_loop

exit_transpose_rows_loop:
  ; Increase column index and iterate
  inc r13
  cmp r13, rsi
  jne transpose_cols_loop

  pop r15
  pop r14
  pop r13
  pop r12
  pop rbx
  pop rbp
```

```
  ret
```

La rutina itera sobre las columnas de M (bucle `transpose_cols_loop`, índice j) y las filas de M (bucle `transpose_rows_loop`, índice i) de a 4 filas por vez. Para cada coordenada (i, j) en M invoca la rutina `transpose_ij`, que copia los coeficientes $M_{i...i+3,j}$ en $M^t_{j,i...i+3}$.

En los casos donde restan 1, 2 o 3 filas, retrocede el índice i de manera que apunte a las últimas 4 filas de M y luego invoca `transpose_ij`. Esto pisa algunos coeficientes ya transpuestos en M^t, pero no es un problema ya que `transpose_ij` sólo lee coeficientes de M, que no son modificados en ningún momento.

A continuación se explica la rutina `transpose_ij`.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Procedure transpose_ij copies m[i..i+3, j] to n[j, i..i+3].     ;;
;; Expects rdi=rows, rsi=cols, rdx=pointer to m, rcx=pointer to n, ;;
;; r12=i, r13=j.                                                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

transpose_ij:

  ; Compute offset for m(i, j)
  mov rax, r12             ; rax = i
  push rdx
  mul rsi                  ; rax = i * cols
  add rax, r13             ; rax = (i * cols) + j
  mov rbx, 4               ; rbx = 4
  mul rbx                  ; rax = 4 * ((i * cols) + j)
  pop rdx

  ; Store m(i, j) in xmm0[0]
  movss xmm0, [rdx + rax]  ; xmm0 = 00000000  | 00000000 | 00000000 | m(i, j)

  ; Compute offset for m(i + 1, j)
  mov rax, r12             ; rax = i
  inc rax                  ; rax = i + 1
  push rdx
  mul rsi                  ; rax = (i + 1) * cols
  add rax, r13             ; rax = ((i + 1) * cols) + j
  mov rbx, 4               ; rbx = 4
  mul rbx                  ; rax = 4 * (((i + 1) * cols) + j)
  pop rdx

  ; Store m(i + 1, j) in xmm1[1]
  movss xmm1, [rdx + rax]  ; xmm1 = 00000000 | 00000000 | 00000000 | m(i+1,j)
  pshufd xmm1, xmm1, 0xE1  ; xmm1 = 00000000 | 00000000 | m(i+1,j) | 00000000
                           ; order = 11 10 00 01

  ; Compute offset for m(i + 2, j)
  mov rax, r12             ; rax = i
  add rax, 2               ; rax = i + 2
  push rdx
  mul rsi                  ; rax = (i + 2) * cols
  add rax, r13             ; rax = ((i + 2) * cols) + j
  mov rbx, 4               ; rbx = 4
  mul rbx                  ; rax = 4 * (((i + 2) * cols) + j)
  pop rdx

  ; Store m(i + 2, j) in xmm2[2]
  movss xmm2, [rdx + rax]  ; xmm2 = 00000000 | 00000000 | 00000000 | m(i+2,j)
```

```
pshufd xmm2, xmm2, 0xC6  ; xmm2 = 00000000 | m(i+2,j) | 00000000 | 00000000
                         ; order = 11 00 01 10

; Compute offset for m(i + 3, j)
mov rax, r12             ; rax = i
add rax, 3               ; rax = i + 3
push rdx
mul rsi                  ; rax = (i + 3) * cols
add rax, r13             ; rax = ((i + 3) * cols) + j
mov rbx, 4               ; rbx = 4
mul rbx                  ; rax = 4 * (((i + 3) * cols) + j)
pop rdx

; Store m(i + 3, j) in xmm3[3]
movss xmm3, [rdx + rax]  ; xmm3 = 00000000 | 00000000 | 00000000 | m(i+3,j)
pshufd xmm3, xmm3, 0x27  ; xmm3 = m(i+3,j) | 00000000 | 00000000 | 00000000
                         ; order = 00 10 01 11

; Merge values in one single XMM register
addps xmm0, xmm1         ; xmm0 = 00000000 | 00000000 | m(i+1,j) |   m(i,j)
addps xmm0, xmm2         ; xmm0 = 00000000 | m(i+2,j) | m(i+1,j) |   m(i,j)
addps xmm0, xmm3         ; xmm0 = m(i+3,j) | m(i+2,j) | m(i+1,j) |   m(i,j)

; Compute offset for n(j, i)
mov rax, r13             ; rax = j
push rdx
mul rdi                  ; rax = j * rows
add rax, r12             ; rax = (j * rows) + i
mov rbx, 4               ; rbx = 4
mul rbx                  ; rax = 4 * ((j * rows) + i)
pop rdx

; Copy contents of xmm0 to n[j, i..i+3]
movups [rcx + rax], xmm0

; End of procedure
ret
```

La rutina `transpose_ij` realiza los siguientes pasos:

1. se guarda $M_{i,j}$ en los bits $0 \dots 31$ de `xmm0`,
2. se guarda $M_{i+1,j}$ en los bits $32 \dots 63$ de `xmm1`,
3. se guarda $M_{i+2,j}$ en los bits $64 \dots 95$ de `xmm2`,
4. se guarda $M_{i+3,j}$ en los bits $96 \dots 120$ de `xmm3`,
5. todos los demás bits de dichos registros se ponen en 0,
6. se suman `xmm0`, `xmm1`, `xmm2` y `xmm3` coordenada a coordenada y se guarda el resultado en `xmm0`, de manera de obtener `xmm0` $= M_{i \dots i+3,j}$,
7. se computa el desplazamiento para (j, i) en `n` (arreglo correspondiente a M^t),
8. se vuelca allí el contenido de `xmm0`, de manera que `n[j, i..i+3]` $= M_{i \dots i+3,j}$.

Esto concluye la rutina `simd_matrix_transpose`.

5.4. EigenMatrix

La clase `EigenMatrix` actúa como un envoltorio alrededor de la clase `Eigen::MatrixXf` de la librería Eigen:

```
class EigenMatrix : public BaseMatrix<EigenMatrix> {
  ...

 private:
  Eigen::MatrixXf m_;
};
```

Todas las operaciones que esta clase implementa son delegadas a las operaciones equivalentes en `Eigen::MatrixXf`. El siguiente ejemplo corresponde a la operación de suma entre matrices:

```
  void EigenMatrix::operator+=(const EigenMatrix& other) {
    m_ += other.m_;
  }
```

El resto de las operaciones se delegan al método correspondiente en la clase `Eigen::MatrixXf` de forma similar.

6. Experimentos

El objetivo de los experimentos es comparar la performance de las distintas implementaciones de la clase `BaseMatrix` en la tarea de entrenar la red neuronal de reconocimiento de dígitos manuscritos durante 100 épocas de entrenamiento.

Cada una de las tres implementaciones (`NaiveMatrix`, `SimdMatrix`, `eigenMatrix`) se compila con niveles de optimización O0, O1, O2 y O3, produciendo 4 grupos de experimentos donde se comparan las tres implementaciones entre sí compiladas con el mismo nivel de optimización.

Para cada grupo de experimentos se mide:

- Tiempo total de entrenamiento.
- Tiempo promedio por época de entrenamiento.

En la siguiente sección se presentan los resultados de dichos experimentos.

7. Resultados

A continuación se presentan los resultados obtenidos.

7.1. Tiempo total de entrenamiento

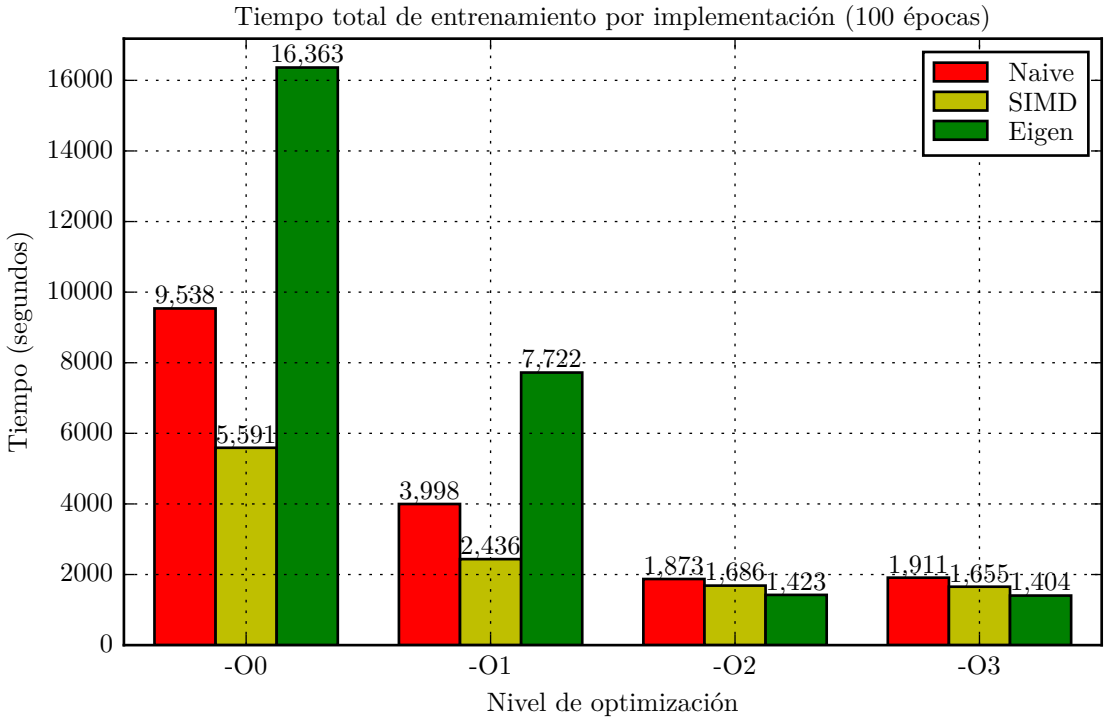

Figura 9: Tiempo total de entrenamiento

7.2. Tiempo promedio por época de entrenamiento

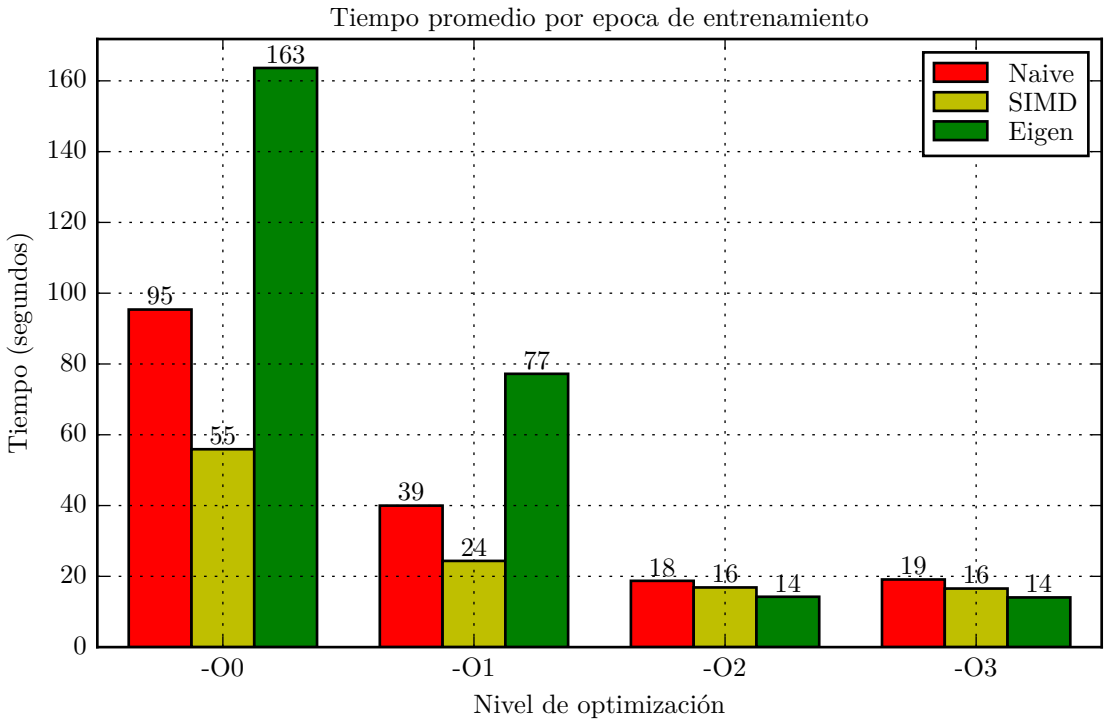

Figura 10: Tiempo promedio por época de entrenamiento

8. Discusión

Analizando las figuras 9 y 10 se puede observar que la implementación `NaiveMatrix` es notablemente más lenta que `SimdMatrix` cuando se deshabilitan las optimizaciones del compilador (nivel de optimización -O0). La diferencia relativa se vuelve cada vez más pequeña en la medida que se incrementa el nivel de optimización, pero en todos los casos `SimdMatrix` demora menos tiempo que `NaiveMatrix`.

En cuanto a `EigenMatrix`, se observa un pésimo desempeño sin optimización del compilador y con el primer nivel de optimización (-O0 y -O1, respectivamente). En ambos casos demora más que `NaiveMatrix`. Esta situación se revierte con los dos niveles de optimización más elevados (-O2 y -O3), en los que `EigenMatrix` es la implementación que mejor se desempeña de las tres.

9. Conclusiones

A partir de lo anterior se concluye que, para la aplicación estudiada en este trabajo, las rutinas escritas en assembler aprovechando las instrucciones SIMD del procesador mejoran notablemente el desempeño del programa comparado a una implementación naive en C++. Sin embargo, estos beneficios se vuelven casi despreciables cuando se compila el mismo programa C++ con niveles de optimización más altos.

Además, se puede concluir que para este tipo de aplicaciones no tiene demasiado sentido el implementar algoritmos de álgebra lineal desde cero, cuando ya existen excelentes librerías de código abierto que ofrecen implementaciones bien implementadas y testeadas.

10. Apéndice: GUI

Con el motivo de depurar la red neuronal implementada en este trabajo, se desarrolló una interfaz gráfica que permite dibujar dígitos con el mouse y visualizar las probabilidades asignadas por la red neuronal para cada dígito del 0 al 9. La misma se ilustra en la figura 11 y se puede utilizar desde un navegador web entrando a `http://leandro.me/Orga2-TPFinal`.

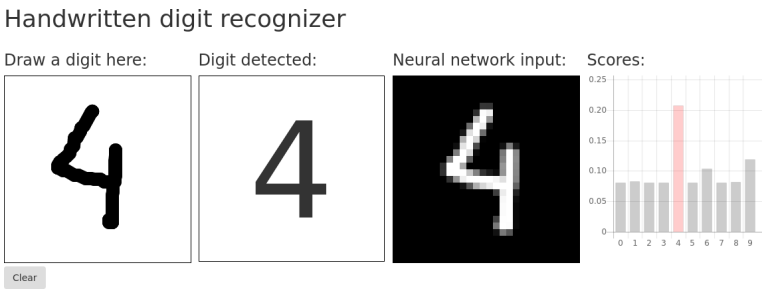

Figura 11: Interfaz gráfica (captura de pantalla)

Para el desarrollo de esta visualización se utilizó Emscripten[15] , un compilador basado en LLVM[16] que transcompila código C++ en JavaScript, permitiendo ejecutar un programa C++ en cualquier navegador web moderno.

La implementación transcompilada a JavaScript es `NaiveMatrix`, ya que Emscripten sólo soporta código C++, y esto imposibilita usar la implementación en assembler.

[15] http://emscripten.org
[16] http://llvm.org/