Redes Neuronales Convolucionales usando GPU

Programación Multinúcleo

Proyecto Final

28 de Noviembre de 2018

Víctor Rendón Suárez

A01022462

Omar Sanseviero Güezmes

A01021626

Introducción

En este proyecto se implementa una red neuronal convolucional (CNN) para la clasificación de dígitos. El objetivo de este proyecto es hacer la implementación con Python, detectar áreas de oportunidad para la paralelización en GPU y contrastar las diferencias entre las implementaciones.

Gran parte del proyecto se desenvolvió en la implementación de la red neuronal utilizando Python y NumPy. Por un lado se utilizó Keras con TensorFlow como backend y por otro la implementación desde cero. El runtime de CPU tiene dos procesadores Intel Xeon, mientras que el GPU es un Tesla K80 con 2496 CUDA cores. Se utiliza Google Colaboratory para la ejecución de los ejemplos.

Redes Neuronales Convolucionales

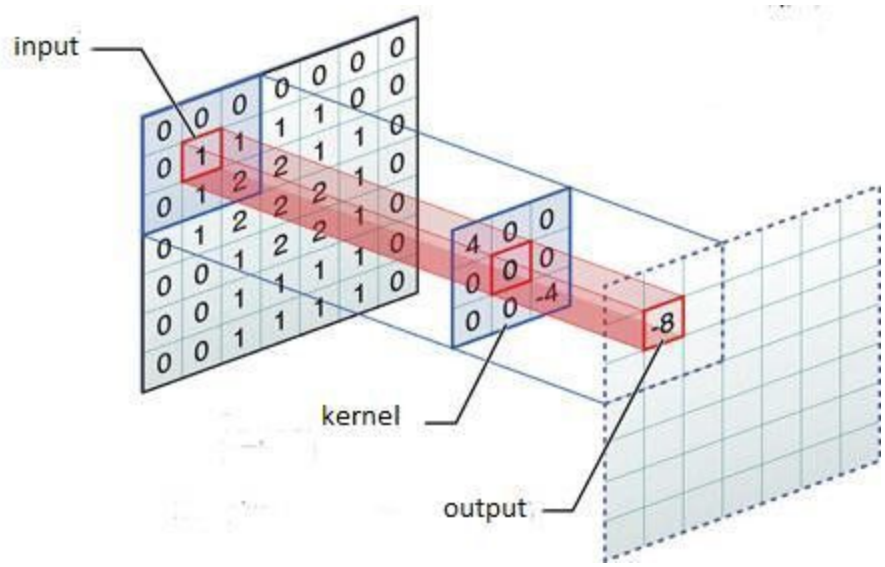
Introducción

En 1988, Yann LeCun presentó el primer proyecto de donde se utilizó backpropagation para entrenar los pesos de los filtros para la clasificación de dígitos[1]. Su trabajo en redes convolucionales culminó en 1998 con la implementación de la primera red neuronal convolucional (CNN), la cual clasificaba dígitos escritos a mano[2]. En el 2012 se utilizaron las CNNs para el reto ImageNet, consiguiendo así los mejores resultados. Las técnicas actuales estado de arte suelen desempeñar mejor que los humanos para la mayoría de las tareas de este tipo (CIFAR, ImageNet, MNIST, etc).

Redes Neuronales Convolucionales

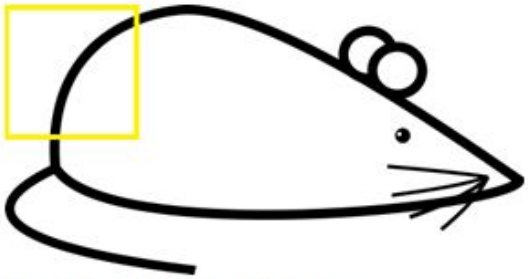
Las CNNs utilizan filtros/kernels para recorrer la imagen y detectar patrones. Lo interesante de estas redes es que los valores de los filtros no se eligen a mano, como se hace para algunas técnicas de visión por computadora, sino que estos aprenden por su propia cuenta. Los filtros simplemente son matrices donde cada elemento es un peso. Al pasar un filtro por la imagen se multiplica cada elemento del filtro por cada elemento correspondiente en la imagen y luego se suma el resultado.

[1] http://yann.lecun.com/exdb/publis/pdf/lecun-89e.pdf
[2] http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf



Asociando esto con neuronas reales, cuando un filtro para detectar cierto patrón lo encuentra en la imagen, la neurona se encenderá más fuerte o, equivalentemente, tendrá valores más altos. Esto permite que cada filtro detecta un feature diferente y el uso de varias capas convolucionales permita identificar patrones más complejos.

Original image



Visualization of the filter on the image

Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

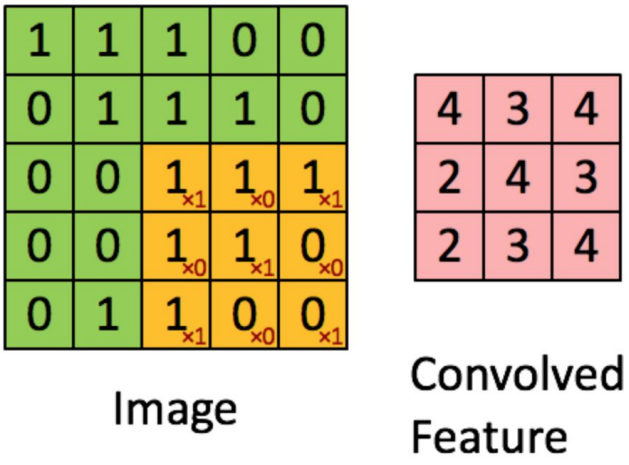
*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

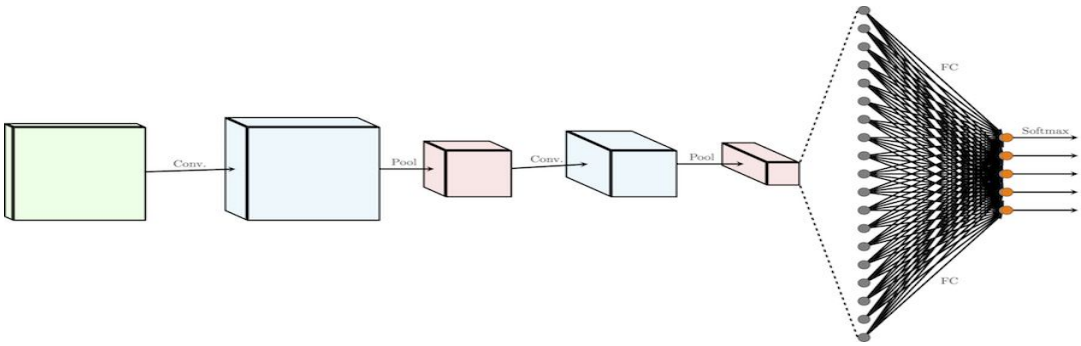
Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

Al pasar el filtro por toda la imagen, se genera una matriz de salida, la cual contiene el resultado del filtro en cada parte de la imagen. Cuando se tienen muchos filtros, se puede ver la salida como una matriz multidimensional, aumentando así la profundidad.

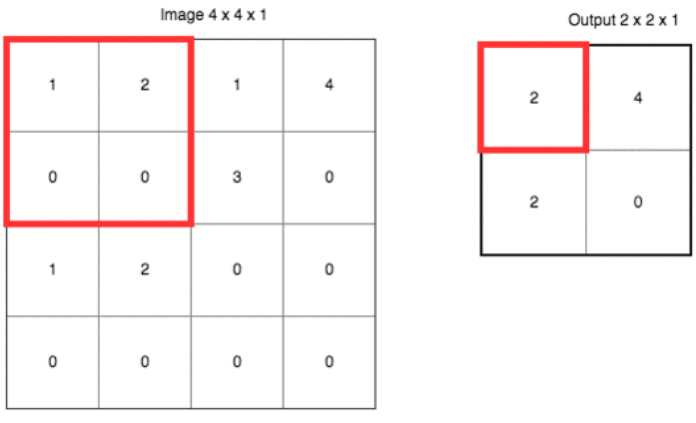


El filtro se va recorriendo con un stride por la imagen. El stride determina cuántos píxeles se mueve el filtro por cada paso. Cuando hay más de un filtro por capa, el output se genera sobre el último eje, creando una "imagen" de 3 dimensiones (para imágenes en blanco y negro, las cuales sólo tienen un canal). La red neuronal convolucional se debe aplanar en su última capa y se conecta con una red neuronal convencional, la cual se encarga de hacer la clasificación normal. La red neuronal convencional se puede ver como una convolucional con un kernel de 1x1 donde cada neurona es un filtro. Esto permite generalizar las matemáticas. El resultado de la última capa debe predecir las probabilidades de salida.



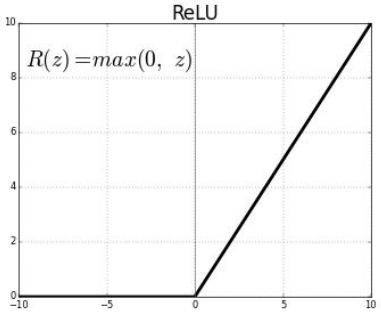
Pooling

Una técnica común es utilizar downsampling para reducir las dimensiones de las imágenes. Esto reduce la memoria necesaria y acelera el entrenamiento. Hay dos técnicas de pooling muy populares, average pooling (promedio) y max pooling. La segunda es más común y tiene una implementación sencilla. Se utiliza una ventana (window) que



recorre la imagen de la misma manera que los filtros. En cada paso (recorriendo según el stride definido), se extrae el valor más alto de la imagen. Además de mejorar los tiempos de entrenamiento y reducir la memoria necesaria, max pooling hace que la red neuronal se enfoque en unas pocas neuronas, lo que funciona como una técnica de regularización (equivalente a utilizar Dropout en una red neuronal normal).

Funciones de Activación



ReLU

La salida de las capas convolucionales y normales se debe activar con una función no linear. Una sencilla de implementar y que logra excelentes resultados es Rectified Linear Unit (ReLU). Esto permite introducir no-linearidad al modelo.

Softmax

La salida de toda la red neuronal debe dar la probabilidad de cada dígito.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

Categorical Cross Entropy

Para algoritmos de entrenamiento se necesita utilizar una función de pérdida (loss functions). La función de pérdida debe decir qué tan correctas son las predicciones. Cuando es un problema de de clasificación multiclase, se utiliza la función de entropía cruzada categorías.

$$H(y, \hat{y}) = \sum_i y_i \log \frac{1}{\hat{y}_i} = -\sum_i y_i \log \hat{y}_i$$

Optimización con Adam

Adam es una extensión del Stochastic Gradient Descent. Adam es un algoritmo de optimización para actualizar los pesos de la red neuronal utilizando información de entrenamiento. Es sencillo de implementar y eficiente. Combina los beneficios de Adaptive Gradient Descent (AdaGrad) y Root Mean Square Propagation (RMSProp). Los parámetros de Adam son:

- alpha: funciona como el learning rate o el step size.
- beta1: El exponential decay rate del primer momento, normalmente 0.99
- beta2: El exponential decay rate del primer momento, normalmente 0.999
- epsilon: Debe ser un valor cercano a cero para prevenir divisiones entre cero.

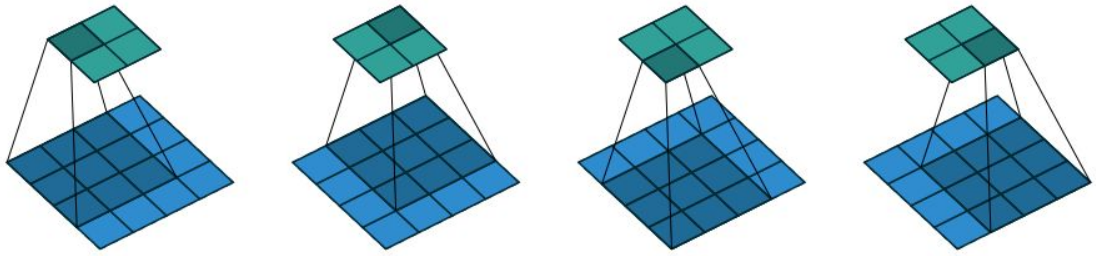
Por default, Keras utiliza

- alpha: 0.001
- beta_1: 0.9
- beta_2: 0.999
- epsilon: 1e-08

Matemáticas

La aplicación (convolución) de un kernel de kxk en una imagen de C canales.

$$(I * K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \sum_{c=1}^{C} K_{m,n,c} \cdot I_{i+m,j+n,c} + b$$



Para el backpropagation se utiliza la derivada parcial del error respecto a los pesos. El peso se propaga desde el final, donde se calcula el error con categorical cross entropy, hacia el inicio.

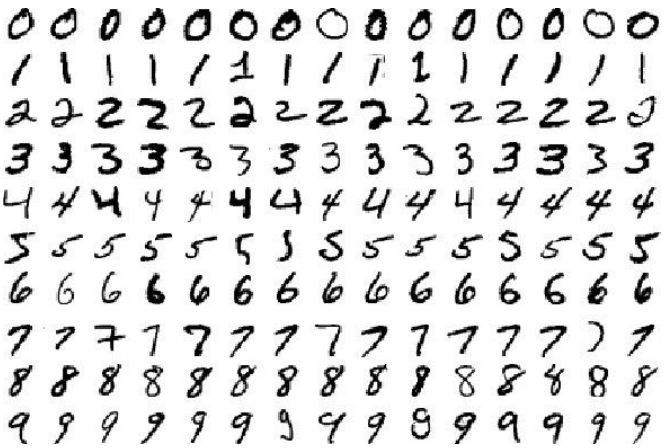
Para Adam se siguió el siguiente algoritmo:

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Datos

Para hacer las pruebas se utilizará una CNN para clasificar dígitos (0-9) del dataset MNIST. Este dataset contiene imágenes en blanco y negro de 28x28 pixeles. La implementación de clasificación con una red neuronal normal suele tardar cientas de veces más que con el uso de convolucionales, por lo que es una excelente oportunidad para probar. Para simplificación del código, sólo se utilizó un training y un testing set. El training set contiene 50,000 elementos y el testing set 10,000.



Los resultados estado del arte como DropConnect[3] y otros conocidos logran resultados a partir de 99.22% de accuracy. Hay que notar que algunas de las soluciones estado de arte utilizan técnicas de aumentación de data, lo cual permite conseguir mejores resultados de los esperados.

Arquitectura

Se utiliza la misma arquitectura en Keras y en la implementación desde cero para lograr resultados consistentes comparables. La red está compuesta por la parte convolucional y por la parte lineal. Las capas convolucionales reciben imágenes de 28x28 y utiliza 4 filtros de 2x2 y un stride de 1. Ambas capas utilizan ReLU como función de activación. Después de las dos capas convolucionales se utiliza una capa de pooling de 4x4 con la técnica Max Pooling para hacer el downsampling con un stride de 4. Este stride es algo alto, pero reduce considerablemente el número de parámetros entrenables.

Una capa de flattening lo aplana a una lista de 196 valores que son recibidos por una capa densa completamente conectada de 128 valores, activada con ReLU, la cual está posteriormente conectada con una de 10 valores. La segunda capa densa se activa con softmax para conseguir las distribuciones de probabilidad de predicción.

Se utiliza la función categorical crossentropy para el cálculo de pérdida, Adam como optimizador y reporte la métrica accuracy.

Se utiliza un batch_size de 32, lo cual permite paralelizar tanto en CPU como en GPU los ejemplos de entrenamiento. Por el costo computacional de la tarea, sólo se ejecuta un epoch (una iteración sobre todos los datos), lo cual consigue resultados suficientemente decentes.

[3] http://cs.nyu.edu/~wanli/dropc/

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 28, 28, 4)         20
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 28, 28, 4)         68
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 4)           0
_________________________________________________________________
flatten_2 (Flatten)          (None, 196)               0
_________________________________________________________________
dense_3 (Dense)              (None, 128)               25216
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1290
=================================================================
Total params: 26,594
Trainable params: 26,594
Non-trainable params: 0
```

Paralelización

Para implementar la paralelización se exploraron diferentes herramientas. La primera de estas fue Numba. Numba es una librería que permite optimizar código de Python y NumPy a código máquina con un just-in-time compiler (jit). Dentro de sus capacidades, está Numba CUDA. En teoría, Numba CUDA te permite escribir kernels en Python como funciones normales, teniendo como limitantes que estos no pueden regresar valores.

En nuestros primeros intentos nos encontramos que Numba tiene muchas limitantes, incluyendo que las versiones de Numba no son compatibles con el CUDA que utiliza PyCUDA. Se utilizó primero jit para hacer optimización en CPU logrando casi reducir a la mitad el tiempo de entrenamiento. En GPU nos encontramos con muchos problemas por las limitantes que tiene Numba para soporte de arreglos de NumPy, aunque en teoría fue diseñado para mantener la misma interfaz. Por ejemplo, no tiene numpy max, por lo que tuvimos que hacer una implementación desde cero para max pooling.

```
@cuda.jit
def maxpool(image, f=2, s=1, pooled=None):
  n_c, h_prev, w_prev = image.shape
  for i in range(n_c):
    curr_y = 0
    out_y = 0
    while curr_y + f <= h_prev:
      curr_x = 0
      out_x = 0

      while curr_x + f <= w_prev:
        max_value = 0
        for col in range(curr_y, curr_y+f):
          for row in range(curr_x, curr_x+f):
            if image[i, col, row] > max_value:
              pooled[i, out_y, out_x] = image[i, col, row]
```

```
                max_value = image[i, col, row]

        curr_x += s
        out_x += 1
    curr_y += s
    out_y += 1
```

Aún así, nos encontramos con problemas con Numba por los cambios de dimensiones constante en nuestros problemas y muchas limitaciones en el soporte de NumPy. En la llamada al kernel se debe definir el número de bloques y el número de threads por bloque.

En la siguiente implementación probamos con PyCUDA. La primera versión utilizó gpuarrays. Al igual que en Numba, nos encontramos con varias limitaciones de soporte de las operaciones de NumPy con matrices de muchas dimensiones. Uno de los mayores bottlenecks en el feedforward es la parte densa, pues ahí se encuentra la mayoría de los parámetros de toda la red neuronal. Utilizamos gpuarrays para la evaluación y logramos reducir de 15 a 10 minutos. Nótese que para la evaluación sólo se hace el feedforward pues sólo se hacen predicciones. Intentamos hacer lo mismo para el entrenamiento, pero, dado que en varios puntos se tuvo que convertir de gpuarray a numpy.array, se crearon bottlenecks que lograron hacer hasta 20 veces más lento el entrenamiento.

La última propuesta fue utilizar PyCUDA pero, en vez de utilizar gpuarrays, escribir kernels en C. Las 4 funciones que propusimos para paralelizar son las siguientes:

- convolution: La cual aplica la convolución de un filtro sobre una imagen.
- maxpool: Aplica max pooling para reducir las dimensiones.
- convolutionBackwards: Actualiza los filtros a partir de la derivada de la capa convolucional
- maxpoolingBackwards.

Pero por cuestión de implementación, no se logró convertir la funcionalidad de estas a C, pues utilizamos técnicas de Python de slicing de código y funciones de NumPy dentro de estas cuatro funciones.

Keras y paralelización

Keras es una librería que permite construir redes neuronales fácilmente. En este proyecto se utilizó TensorFlow como backend de Keras. En CPU, Keras utiliza todos los threads disponibles, por lo que consigue excelentes resultados. Adicionalmente, utiliza estructuras de datos optimizadas para la matemática de las redes neuronales. Cuando hay GPU, Keras lo utiliza automáticamente, logrando un speedup de 2.

TensorFlow utiliza dos diferentes interfaces para las operaciones en CPU y en GPU: en C++ y CUDA. Por ejemplo, tf.transpose[4] tiene su interfaz definida en C++-

```
REGISTER_OP("Transpose")
    .Input("x: T")
    .Input("perm: Tperm")
    .Output("y: T")
```

[4] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/ops/array_ops.cc#L1345

```
    .Attr("T: type")
    .Attr("Tperm: {int32, int64} = DT_INT32")
    .SetShapeFn(TransposeShapeFn);
```

En la interfaz se especifican las entradas, la salida y la forma del output dado un input. Todos son tensores y permutaciones de tensores. En el header de la operación hay dos operadores definidos[5].

```
class TransposeGpuOp : public TransposeOp { ... }
class TransposeCpuOp : public TransposeOp { ... }
```

Al final, cada definición está vinculada con una interfaz diferente.

```
template <typename T, bool conjugate>
void TransposeSimple(const GPUDevice& d, const Tensor& in,
                     const gtl::ArraySlice<int32> perm, Tensor* out) {
  // Ensures we can use 32-bit index.
  const int64 nelem = in.NumElements();
  CHECK_LT(nelem, kint32max) << "Tensor too large to transpose on GPU";
  // Pack strides and permutation into one buffer.
  const int32 ndims = in.dims();
  gtl::InlinedVector<int32, 24> host_buf(ndims * 3);
  gtl::InlinedVector<int32, 8> in_strides = ComputeStride<int32>(in.shape());
  gtl::InlinedVector<int32, 8> out_strides = ComputeStride<int32>(out->shape());
  // Dimension permutation.
  for (int i = 0; i < ndims; ++i) {
    host_buf[i] = in_strides[i];
    host_buf[ndims + i] = out_strides[i];
    host_buf[ndims * 2 + i] = perm[i];
  }
  // Copies the input strides, output strides and permutation to the device.
  auto num_bytes = sizeof(int64) * host_buf.size();
  auto dev_buf = d.allocate(num_bytes);
  // NOTE: host_buf is not allocated by CudaHostAllocator, and
  // therefore we are doing a sync copy effectively.
  d.memcpyHostToDevice(dev_buf, host_buf.data(), num_bytes);
  // Launch kernel to q[...] = p[...].
  const T* p = reinterpret_cast<const T*>(in.tensor_data().data());
  T* q = reinterpret_cast<T*>(const_cast<char*>((out->tensor_data().data())));
  CudaLaunchConfig cfg = GetCudaLaunchConfig(nelem, d);
  TransposeKernel<T, conjugate>
      <<<cfg.block_count, cfg.thread_per_block, 0, d.stream()>>>(
          cfg.virtual_thread_count, p, reinterpret_cast<const int32*>(dev_buf),
          ndims, q);
  // Safe to deallocate immediately after the kernel launch.
  d.deallocate(dev_buf);
}
```

```
void TransposeSimple(const CPUDevice& device, const Tensor& in,
                     const gtl::ArraySlice<int32> perm, Tensor* out) {
  const int ndims = in.dims();
  gtl::InlinedVector<int64, 8> in_strides = ComputeStride<int64>(in.shape());
  gtl::InlinedVector<int64, 8> out_strides = ComputeStride<int64>(out->shape());
  const T* p = reinterpret_cast<const T*>(in.tensor_data().data());
  T* q = reinterpret_cast<T*>(const_cast<char*>((out->tensor_data().data())));
  auto transpose_fn = [=, &in_strides, &out_strides, &perm](int64 begin,
                                                            int64 end) {
    for (int64 o_idx = begin; o_idx < end; ++o_idx) {
      int64 i_idx = 0;
      int64 t = o_idx;
      for (int i = 0; i < ndims; ++i) {
        const int64 ratio = t / out_strides[i];
        t -= ratio * out_strides[i];
        i_idx += ratio * in_strides[perm[i]];
      }
      if (conjugate) {
        q[o_idx] = Eigen::numext::conj(p[i_idx]);
      } else {
        q[o_idx] = p[i_idx];
      }
    }
  };
  double cycles_per_element =
      (conjugate ? 1 : 0) + ndims * (Eigen::TensorOpCost::DivCost<int64>() +
                                     2 * Eigen::TensorOpCost::MulCost<int64>() +
                                     2 * Eigen::TensorOpCost::AddCost<int64>());
  Eigen::TensorOpCost cost(/*bytes_loaded=*/sizeof(T),
                           /*bytes_stored=*/sizeof(T), cycles_per_element);
  device.parallelFor(in.NumElements(), cost, std::move(transpose_fn));
}
```

Resultado

	Accuracy	**Tiempo training**
CPU Normal	91.01%	2:02:21
CPU optimizado con Numba	-	1:17:00

[5] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/transpose_op.h

(just-int-time compiling)		
CPU Keras	94.85%	38 segundos
GPU Keras	94.91%	16 segundos

Conclusiones

En este proyecto logramos ver que el uso de GPU logra un speedup de dos utilizando Keras. La implementación de la red neuronal convolucional no resultó ser sencilla y requirió revisar numerosos recursos como papers y tutoriales. En la etapa de evaluación, nuestra versión en CPU se tarda varios minutos (lo cual sólo es el feedforward), e incluso paralelizando con GPU arrays, lo cual logra un speedup de 1.5, el tiempo es de 10 minutos vs unos segundos utilizando Keras. Esto nos hace pensar que Keras utiliza estructuras de datos altamente optimizadas a bajo nivel para la representación de la información y las operaciones entre los tensores. El uso excesivo de GPU arrays resultó ser contraproducente pues requería pasar memoria de host a device, agregando puntos de bottlenecks y alentando el algoritmo de entrenamiento.

Algo que es importante destacar es que utilizamos Google Colaboratory, lo cual hace que los resultados no sean tan estables, pues hay veces que puedes tener 5% del GPU o 100% del GPU, según el número de otros usuarios compartiendo.

Fuentes

- https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/
- https://medium.com/the-bioinformatics-press/only-numpy-understanding-back-propagation-for-max-pooling-layer-in-multi-layer-cnn-with-example-f7be891ee4b4
- https://arxiv.org/abs/1412.6980
- https://towardsdatascience.com/convolutional-neural-networks-from-the-ground-up-c67bb41454e1
- https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
- http://cs231n.github.io/convolutional-networks/#conv
- http://numba.pydata.org/
- https://medium.com/@erikhallstrm/backpropagation-from-the-beginning-77356edf427d
- https://arxiv.org/pdf/1603.07285.pdf
- https://pdfs.semanticscholar.org/5d79/11c93ddcb34cac088d99bd0cae9124e5dcd1.pdf