Pasantía Industrial

Ingeniería de Sistemas

Presentado ante la ilustre Universidad de Los Andes

Perfilado temporal en redes neuronales convolucionales

Por

Br. Sarait de Jesús Hernández Ruiz

Tutor Industrial: Ing. Luis Portillo

Octubre, 2018

©2018 Universidad de Los Andes Mérida, Venezuela

Perfilado temporal en redes neuronales convolucionales

Br. Sarait de Jesús Hernández Ruiz

Pasantía Industrial

Ingeniería de Sistemas — Sistemas Computacionales, 42 páginas

Resumen: En la clasificación de imagenes, las redes neuronales convolucionales son altamente populares, ya que permiten redes más profundas, sin tener un número excesivo de parámetros que aprender por la red. Por otra parte, el proceso de inferencia consiste en evaluar la presión de la red sobre entradas desconocidas. El proceso de perfilado temporal, consiste en medir el tiempo de ejecución y el uso de memoria en las operaciones involucradas en un algoritmo, este proceso será aplicado a las CNNs (convolutional neural networks, por sus siglas en inglés), usando como framework de desarrollo TensorFlow. La empresa en la que se desarrolló el presente proyecto de pasantía industrial, Merida Technology Group, C.A. (MeridaTech), ubicada en el C.C. Alto Prado, Local 83, es un espacio en el que abunda el desarrollo y la innovación, con vista en dar un lugar a las más grandes mentes de la computación para poner en alto el nombre de Venezuela y América Latina en la competencia mundial, es así como lleva cuatro años en el mercado internacional de forma exitosa.

Palabras clave: Redes neuronales convolucionales, TensorFlow, perfilado temporal, MeridaTech

Índice

Índice de Tablas **v**

Índice de Figuras **vi**

1 Introducción **1**

1.1 El espacio de desarrollo 1

1.2 Estructura del documento 2

2 MeridaTech **3**

3 Plan de trabajo **4**

4 El proceso de desarrollo **6**

4.1 Redes Neuronales Convolucionales en la clasificación de imágenes . . . 6

4.1.1 Arquitectura ConvNet 7

4.1.2 Entrenamiento 17

4.1.3 Inferencia 18

4.2 TensorFlow como framework de desarrollo 18

4.2.1 Tensor 19

4.2.2 Jerarquía de las capas de abstracción de TensorFlow 19

4.2.3 Representación de la operaciones en TensorFlow 23

4.2.4 Perfilado temporal en TensorFlow 25

4.2.5 Ventajas y desventajas de TensorFlow 26

4.3 Perfilado y optimización 27

4.3.1 Dataset de entrenamiento CIFAR-10 27

4.3.2 Arquitectura e implementación del modelo 27
4.4 Análisis de los cuellos de botella: Perfilado temporal 32
4.5 Propuestas para la mejora del rendimiento 34
4.6 Conocimientos adquiridos durante la carrera vinculados al desarrollo del proyecto 38

5 Conclusiones 40

Bibliografía 41

Índice de Tablas

3.1 Programa de actividades del plan de trabajo de las pasantías industriales 5

4.1 Descripción de los módulos del código 28
4.2 Descripción de la función inference() 29
4.3 Resultados promedio perfilado temporal inferece() 33
4.4 Stride 2: Perfilado temporal en inference(), precisión: 79.6% 34
4.5 Stride 3: Perfilado temporal en inference(), precisión: 74.7% 35
4.6 Stride 5: Perfilado temporal en inference(), precisión: 70% 35
4.7 Reducción de la cantidad de filtros: Perfilado temporal en inference(), precisión: 83.7% 37
4.8 Aumento de pooling: Perfilado temporal en inference(), precisión: 86.3% 38

Índice de Figuras

4.1 Arquitectura de ejemplo ConvNet 7
4.2 Imagen a color con canales RGB 8
4.3 Operación de convolución entre un filtro y una imagen 9
4.4 Convolución imagen - filtro W0, resultando el mapa de activación W0 . 10
4.5 Convolución imagen - filto W1, resultando el mapa de activación W1 . 11
4.6 Funcion ReLU (Unidad Lineal Rectificada) 13
4.7 Operación Max Pooling 14
4.8 Pooling aplicado por mapa de activación 15
4.9 Etapa de clasificación CNNs 16
4.10 Tensores 19
4.11 Jerarquía en TensorFlow 20
4.12 Entrenamiento distribuido 22
4.13 TensorBoard 23
4.14 Ejemplo de DAG 24
4.15 Interfaz gráfica Profiler 25
4.16 Perfilado temporal, resultados del terminal 26
4.17 Arquitectura de la función de inferencia 29
4.18 Resultados perfilado temporal inference() 33

Capítulo 1

Introducción

La evolución de las redes neuronales en los últimos años ha sido tal que ha tenido un gran impacto en la automatización de procesos industriales, en el control de vehículos autoconducidos, en clasificación de imágenes, en reconocimientos de patrones, entre algunas áreas de investigación y desarrollo. En particular, las redes neuronales convolucionales han sido ampliamente usadas en reconocimiento de imágenes y videos, donde la detección, predicción, clasificación, estimación y, en general, técnicas de inferencia estadística basada en registros (imágenes y/o videos) son algunas de las aplicaciones comúnmente encontradas en la literatura. Sin embargo, la implementación de los algoritmos de inferencia (evaluación de la red ante una entrada desconocida) y de entrenamiento en este tipo de redes neuronales es computacionalmente costosa y en general requieren de grandes recursos computacionales y elevados tiempos de ejecución. El objetivo principal del proyecto de pasantías es analizar técnicas de optimización que mejoren la velocidad de la inferencia en redes neuronales convolucionales usando TensorFlow como framework de desarrollo.

1.1 El espacio de desarrollo

MeridaTech ha sido un lugar de crecimiento personal, profesional e intelectual para mí por dos años, cumpliendo un cargo de beca trabajo a medio tiempo, sin embargo, la experiencia como pasante ha sido de lo más enriquecedora, me siento honrada de

pertenecer a una empresa de desarrollo como esta.

En MeridaTech existe un ambiente de trabajo cómodo, activo y dinámico. En un proyecto de pasantías, el aprendizaje no proviene solamente del desarrollo del trabajo, sino también de la interacción y la observación; este tipo de pasantías permite no solo dar un espacio en el cual aplicar los conocimientos adquiridos en el salón de clase, de los libros y la práctica, sino una forma de adquirir nuevos conocimientos y herramientas, pero más importante, dar un primer vistazo a un ambiente laboral más real al que estaba acostumbrada. En MeridaTech no se aprende solo haciendo, sino interactuando, observando y absorbiendo toda la información posible.

1.2 Estructura del documento

Este trabajo se estructura de la siguiente manera:

En el capítulo 2 se describe la empresa en la que se desarrolló el presente proyecto (MeridaTech) y sus principales actividades y especialidades, para luego introducir a su equipo de trabajo y sus planes de desarrollo.

En el capítulo 3 se expone el plan de trabajo original del proceso de pasantías industriales.

El capítulo 4 relata los conceptos básicos de las redes neuronales convolucionales y de tensorflow, para finalizar se analizan los resultados obtenidos de la realización del perfilado temporal sobre algunos experimentos, donde se busca disminuir el costo computacional y el uso de memoria del proceso de inferencia al evaluar la red con entradas desconocidas.

Por último, en el capítulo 5 se presentan las conclusiones del proyecto.

Capítulo 2

MeridaTech

Merida Technology Group (MeridaTech, 2018) es una empresa especializada en el campo de la Computación de Alto Desempeño (HPC, por sus siglas en inglés, High-Performance Computing), dedicada al desarrollo de software científico de calidad industrial y al soporte en investigación de clase mundial. MeridaTech reúne a un grupo de grandes profesionales de la computación, ingeniería eléctrica y áreas afines para resolver grandes problemas en el campo de la aceleración de software y el diseño e implementación de algoritmos innovadores, persiguiendo también la meta de incluir a los países de América Latina como sólidos competidores en la Ingeniería de Software a nivel mundial.

Algunos de los problemas que se empeñan en resolver en MeridaTech son:

- Diseñar e implementar algoritmos innovadores y de gran calidad.
- Llevar a cabo investigaciones relacionadas con cualquier aspecto de una aplicación y apoyar en la implementación de nuevas ideas y creación de patentes.

Actualmente, MeridaTech cuenta con un espacio de $96m^2$ con condiciones aptas y cómodas para el desarrollo del trabajo día a día, para el cual se hizo una inversión de cerca de 25.000$, y eso solo en infraestructura. Además de eso, cuentan con equipos de la más alta tecnología, necesarios para el desarrollo de los productos de alta calidad para la industria científica.

Capítulo 3

Plan de trabajo

El plan de trabajo propuesto al inicio de las pasantías fue el siguiente:

Objetivo:

Diseño, desarrollo e implementación de algoritmos que optimicen la inferencia en redes neuronales convolucionales.

Objetivos específicos:

- Familiarizarse con los fundamentos de redes neuronales convolucionales y su campo de aplicación en clasificación de imágenes.
- Analizar los cuellos de botellas de tipo computacional, de acceso a memoria o de comunicación que se presentan en el proceso de inferencia (evaluación de la red entrenada ante una entrada desconocida) en estas redes mediante el uso de técnicas de perfilado temporal.
- Proponer arquitecturas que minimicen los cuellos de botella encontrados en el punto anterior.
- Diseñar, desarrollar e implementar algoritmos propios que mejoren el desempeño de inferencia usados en redes neuronales convolucionales

- Elaborar un informe técnico donde se describan la arquitectura usada y los resultados obtenidos.

Plan:

La descripción de las actividades que se realizaron como parte del plan de trabajo y su distribución semanal, se exponen en la Tabla 3.1.

Tabla 3.1: Programa de actividades del plan de trabajo de las pasantías industriales

Semana	**Descripción de la actividad**
1-2	Familiarizarse con los fundamentos de redes neuronales convolucionales y su campo de aplicación en clasificación de imágenes.
2-3	Analizar los cuellos de botellas de tipo computacional, de acceso a memoria o de comunicación que se presentan en el proceso de inferencia (evaluación de la red entrenada ante una entrada desconocida) en estas redes mediante el uso de técnicas de perfilado temporal.
3-4	Proponer arquitecturas que minimicen los cuellos de botella encontrados en la actividad anterior.
5-6	Diseñar, desarrollar e implementar algoritmos propios que mejoren el desempeño del proceso de inferencia en redes neuronales convolucionales.
7-8	Elaborar un informe técnico donde se describan la arquitectura usada y los resultados obtenidos.

Capítulo 4

El proceso de desarrollo

4.1 Redes Neuronales Convolucionales en la clasificación de imágenes

Las redes neuronales convolucionales (ConvNets o CNNs) son muy similares a las redes neuronales normales. Están formadas por neuronas que tienen pesos y sesgos que se pueden aprender (conocidos como parámetros libres). Cada neurona recibe algunas entradas, realiza un producto punto de dicha entrada y su vector de pesos y suma el sesgo, opcionalmente lo sigue con una función de activación para agregar la no linealidad a la red, y por último se calcula en la última capa la función de pérdida. Sin embargo, las redes neuronales normales no se adaptan bien a la clasificación imágenes. Por ejemplo: en CIFAR-10 (el dataset de estudio), las imágenes son de tamaño $32\times32\times3$ (32 de ancho, 32 de alto, 3 canales de color), por lo que una sola neurona totalmente conectada en una primera capa oculta de una red neuronal normal tendría $32 \times 32 \times 3 + 1 = 3073$ parámetros (pesos). Esta cantidad parece manejable, pero es evidente que esta estructura completamente conectada no se adapta a imágenes más grandes. Por otra parte, es casi seguro que la red está compuesta por muchas de esas neuronas, por lo que la cantidad de parámetros crecería muy rápidamente. Claramente, esta conectividad completa es un desperdicio y la gran cantidad de parámetros originaría rápidamente un sobreajuste o overfitting (efecto de sobreentrenar una red ajustandola a características muy específicas, ocasionando fallas en el proceso de inferencia) en la red,

(Fei Fei Li, 2017). Por lo contrario, las arquitecturas ConvNet tratan las entradas como volúmenes (imágenes), estas poseen ancho, alto y profundidad y se encargan de explotar las propiedades de la entrada. La convolución reduce la cantidad de parámetros libres y permite que la red sea más profunda con menos parámetros.

4.1.1 Arquitectura ConvNet

Una CNN consiste en una serie de capas convolucionales y (opcionalmente) de pooling seguidas por capas completamente conectadas (Fei Fei Li, 2017). La entrada de una capa convolucional es un volumen de datos, con ancho, alto y profundidad (la palabra profundidad aquí se refiere al número de canales de los datos, en principio, la cantidad de canales de la imagen (RGB) y no a la profundidad de una red neuronal completa, que se refiere al número total de capas en una red), esta capa cuenta con K filtros o kernels, con alto y ancho menor al de la entrada y de igual profundidad; cada filtro se convoluciona con el volumen de datos, para producir K mapas de activación, y el conjunto de mapas de activación servirán de entrada para la siguiente capa convolucional. Al final de la arquitectura ConvNet el volumen obtenido es "aplanado", transformando toda la información a un solo vector que contiene las características extraídas de la imagen, vector que será la entrada para la capa totalmente conectada que está encargada de hacer la clasificación. En la Figura 4.1 se muestra una arquitectura ConvNet de ejemplo.

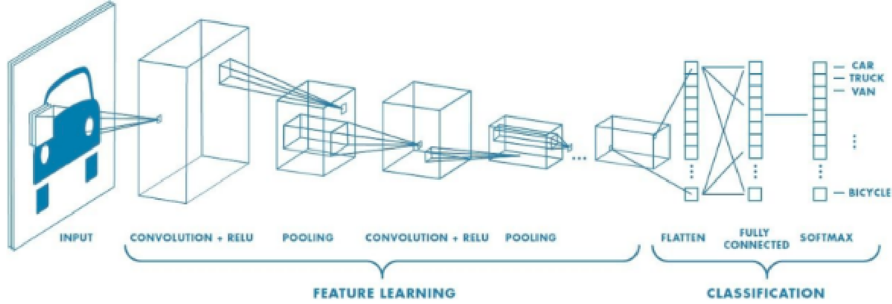

Figura 4.1: Arquitectura de ejemplo ConvNet
Tomada de (Shyamal Patel, 2017)

Capa convolucional

La capa convolucional recibe como entrada imágenes, cada imagen se puede representar como una matriz o conjuntos de matrices de valores de píxeles. El canal es un término convencional utilizado para referirse a un determinado componente de una imagen. Una imagen a color tiene (comúnmente) tres canales: rojo, verde y azul. Cada canal se puede imaginar como una matriz 2D, con valores para cada píxel en el rango de 0 a 255, y la imagen completa como estas tres matrices apiladas una sobre la otra, con tamaño $W \times H \times D$ $(ancho \times alto \times profundidad)$. En la Figura 4.2 se puede observar la representación volumétrica para una imagen con tres canales.

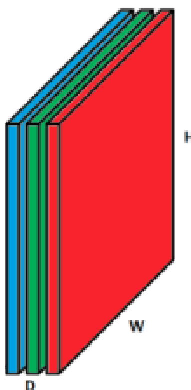

Figura 4.2: Imagen a color con canales RGB
Adaptada de (Fei Fei Li, 2018)

Un filtro o kernel también se puede representar como una matriz cuyas ponderaciones están destinadas a la detección de características (bordes, curvas...). En redes neuronales convolucionales, al igual que la imagen de entrada es volumétrico, usualmente, conserva el número de canales de la entrada, pero con dimensiones de ancho y alto más pequeñas. La convolución es una operación matemática que consiste en hacer un producto punto deslizante, normalmente, entre dos vectores. En CNN la convolución se aplica entre la imagen de entrada y el filtro, el resultado de esta operación es denominado mapa de activación. Por ejemplo, considere una imagen de $5 \times 5 \times 1$ y un filtro de $3 \times 3 \times 1$ (Nótese que se conserva la profundidad del filtro respecto a la imagen) la operación de convolución entre la imagen y el filtro se calcula

como se ve en la Figura 4.3:

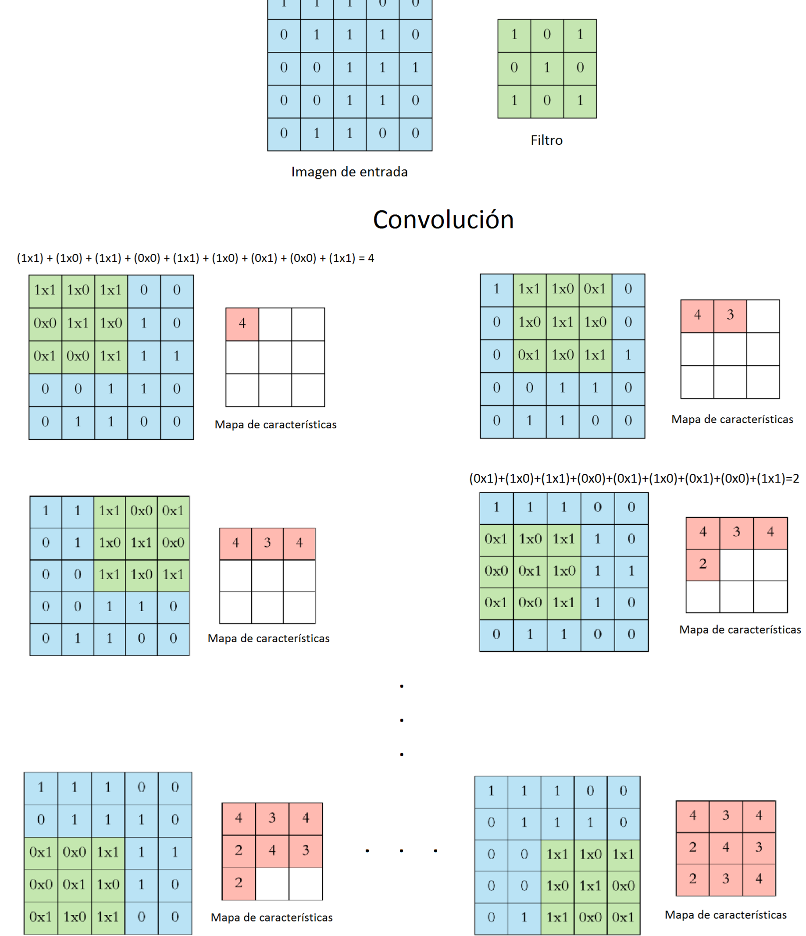

Figura 4.3: Operación de convolución entre un filtro y una imagen
Adaptada de (Dertat, 2017)

Como es de suponer, cada filtro convolucionado sobre la imagen, genera un mapa de activación diferente, y el conjunto de mapas de activación producen el volumen de la salida de la capa convolucional.

Sea una imagen de tamaño $7 \times 7 \times 3$ y dos filtros de tamaño $3 \times 3 \times 3$, el volumen de salida, resultado de la operación de convolución entre la imagen y los filtros se calculan como se observa en las Figuras 4.4 y 4.5:

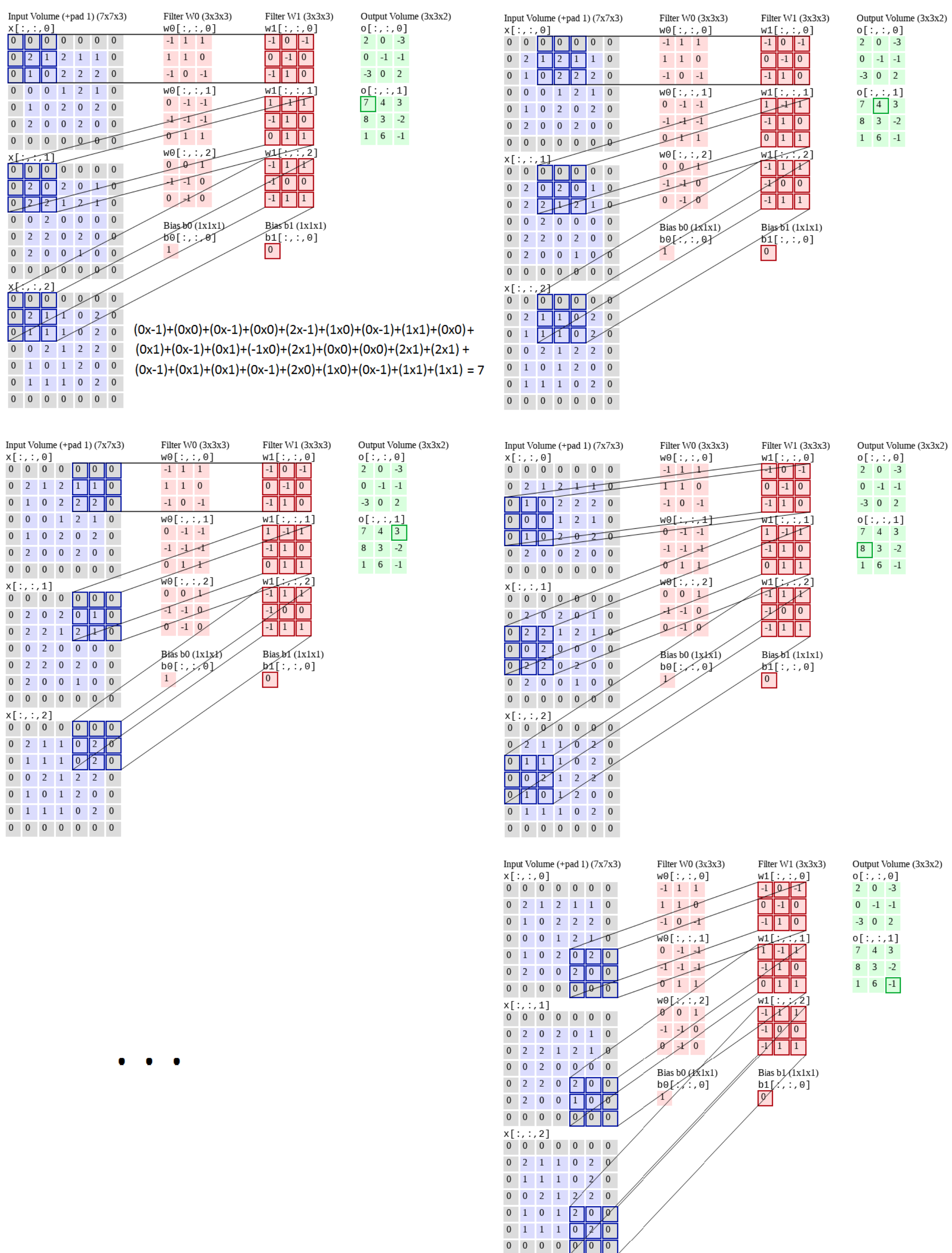

Figura 4.4: Convolución imagen - filtro W0, resultando el mapa de activación W0 Adaptada de (Fei Fei Li, 2017)

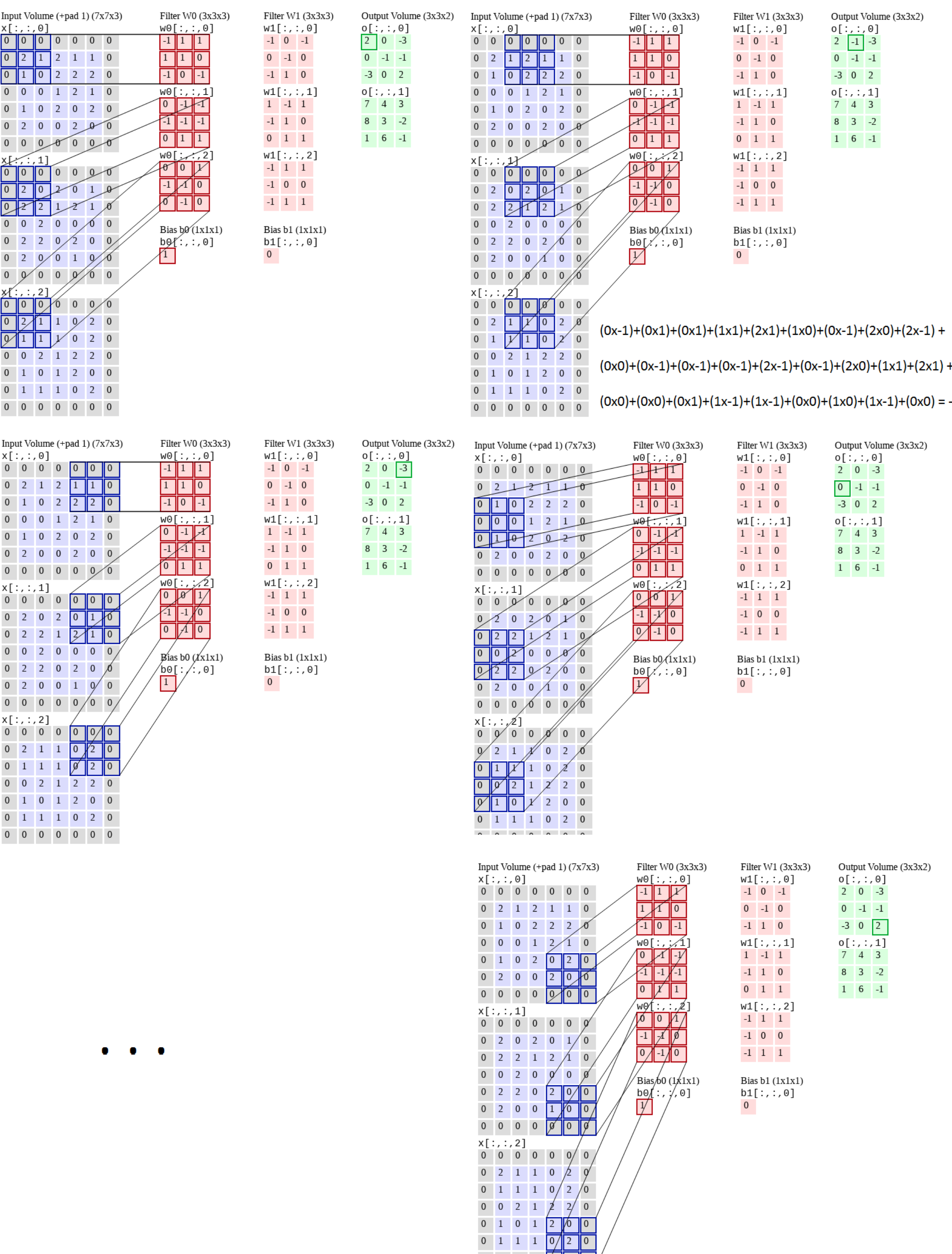

Figura 4.5: Convolución imagen - filto W1, resultando el mapa de activación W1 Adaptada de (Fei Fei Li, 2017)

El objetivo principal de la convolución es extraer características de la imagen de entrada. Cuenta con cuatro hiper parámetros (valores que podemos elegir) (Fei Fei Li, 2017):

- Cantidad de filtros: define la cantidad de canales o mapas de activación que tendrá como profundidad el volumen de salida de la capa convolucional.
- Tamaño de filtro: por lo general, mucho más pequeño que las dimensiones del volumen de entrada, para una mejor detección de características, siendo los tamaños más comunes 2×2, 3×3 o 5×5.
- Stride: como ya fue mencionado, el filtro se desliza sobre la imagen para ser convolucionado con esta, el stride define la cantidad de pixeles que se moverá el filtro sobre la imagen, por ejemplo, si tenemos stride uno el filtro se moverá sobre la imagen de un pixel a la vez (véase la Figura 4.3), si tenemos stride dos el filtro se moverá sobre la imagen de dos píxeles a la vez (véase la Figura 4.4).

  No es difícil notar que dependiendo del tamaño de la imagen, el filtro y el stride, la convolución puede no "cuadrar", para saber si el tamaño del filtro y el stride elegido es adecuado para la imagen vemos la cantidad de productos punto que debemos hacer entre la imagen y el filtro, que obtenemos con la siguiente fórmula:

  $(N - F)/S + 1$, donde N es la altura (o la anchura) de entrada, F la altura (o anchura) del filtro y S el stride (Ng, 2018). Si esta operación tiene como resultado un número no entero, la combinación del tamaño del filtro y el stride no se puede aplicar sobre el volumen de entrada ya que necesitaríamos una cantidad no entera de productos punto. Por ejemplo, para la Figura 4.3, cuyo tamaño del volumen de entrada es de $7 \times 7 \times 3$ y del filtro es de $3 \times 3 \times 3$, con un stride igual a 2:

  $(N - F)/S + 1 = (7 - 3)/2 + 1 = 3$, esto implica que podemos desplazar el filtro sobre la imagen a lo alto y ancho tres veces, resultando así, el tamaño del mapa de activación 3×3.

  Por otra parte si tenemos un stride de 3: $(N - F)/S + 1 = (7 - 3)/3 + 1 = 2.33$, esto implica que no se puede hacer la convolución con estos valores.

- Padding: El tamaño de la data, con cada convolución se irá reduciendo en altura y anchura, algo que no es siempre conveniente, además, la aplicación del filtro en los bordes de la imagen no es ideal, siempre hay pérdida de información, ya que los pixeles en los bordes nunca quedan en el centro de los filtros, por ejemplo. Ambos problemas pueden ser solventados con el uso de padding, que consiste simplemente en agregar filas y columnas adicionales, normalmente de ceros, alrededor de la imagen (véase la Figura 4.3).

  Al tomar en cuenta el padding, la fórmula anterior queda de la siguiente forma:

  $(N - F + 2P)/S + 1$, donde N, F y S siguen denotando lo mismo y P denota la cantidad de filas (o columnas) que añadimos de padding. De igual forma, la expresión debe tener un resultado entero para que la elección de los hiper parámetros sea válida.

  Es bastante común elegir el padding de tal forma que $(N - F + 2P)/S + 1 = N$, es decir, que las dimensiones de la imagen se mantengan.

ReLU

La ReLU (Unidad lineal rectificada) es una función de activación, aplicada por píxel, que reemplaza todos los valores negativos en el conjunto de mapas de activación por cero (véase la Figura 4.8). El objetivo de la ReLU es introducir la no linealidad en la ConvNet, ya que la mayoría de los datos del mundo real son no lineales (Karn, 2016). Generalmente es aplicada después de cada operación de convolución, por ser una operación lineal (producto punto entre matrices y adiciones).

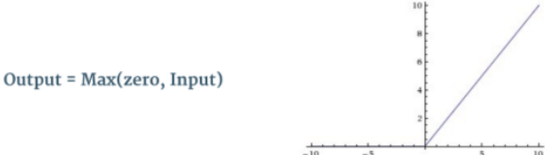

Figura 4.6: Funcion ReLU (Unidad Lineal Rectificada)
Tomada de (Karn, 2016)

Otras funciones no lineales tales como tanh o sigmoide también se pueden utilizar en lugar de ReLU, pero se ha encontrado que para el caso de las ConvNets, ReLU a obtenido los mejores resultados (Ng, 2018).

Capa Pooling

La función de Pooling o submuestreo es reducir progresivamente el tamaño espacial del volumen de entrada (por lo general, el volumen de salida de la capa ReLU), pero conservando la información más importante. Al reducir la cantidad de parámetros y cálculos en la red, controla el sobreajuste.

Consiste en aplicar una función (Max, Average, Sum etc.) sobre un vecindario local de tamaño $N \times N$ en cada mapa de activación del volumen de entrada por separado, para su aplicación, se mantiene el concepto de stride. Por ejemplo, en el caso de Max Pooling, si definimos un vecindario o ventana local de 2×2, tomaríamos el elemento más grande del mapa de activación dentro de ese vecindario (Karn, 2016). En lugar de tomar el elemento más grande, también podríamos tomar el promedio (Average Pooling) o la suma de todos los elementos en esa ventana (Sum Polling). En la práctica, el Max Pooling ha demostrado funcionar mejor. La Figura 4.7 muestra un ejemplo de aplicar Max Pooling sobre un mapa de activación rectificado (resultado de hacer la operación de convolución + ReLU) mediante el uso de una ventana de 2×2 y un stride 2.

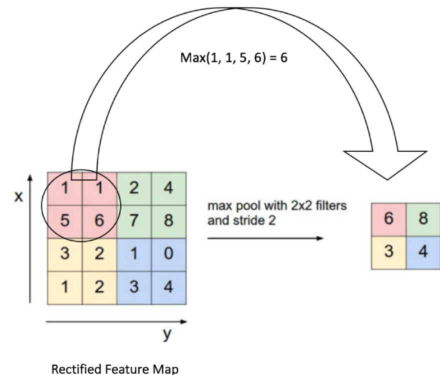

Figura 4.7: Operación Max Pooling

Tomada de (Karn, 2016)

Como ya fue mencionado, el Pooling se aplica por separado a cada mapa de activación, por lo tanto, aunque las dimensiones ancho y alto del volumen de entrada disminuyen, la profundidad se mantiene igual, como se muestra en la Figura 4.8.

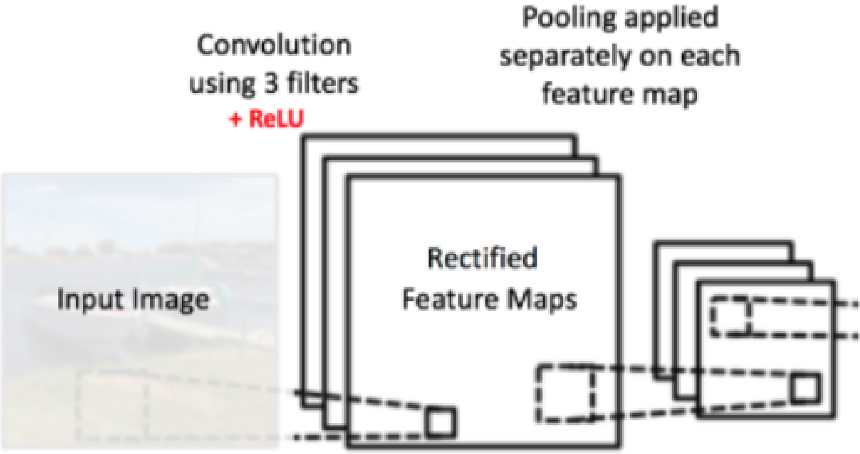

Figura 4.8: Pooling aplicado por mapa de activación
Tomada de (Karn, 2016)

Una ventaja evidente del pooling, es que hace que la red sea invariante para pequeñas transformaciones y distorsiones del volumen de entrada, además nos ayuda a llegar a una representación equivalente a una menor escala del mapa de activación, lo que es muy conveniente ya que podemos detectar objetos en una imagen sin importar dónde se encuentren.

Softmax

Softmax es una función que toma un vector de valores reales y escala dichos valores al intervalo [0,1] de tal forma que la suma de los valores sumen 1. Es muy usada en problemas de clasificación, donde idealmente queremos como salida una distribución de probabilidad (Ng, 2018).

Capa totalmente conectada

La capa totalmente es una red neuronal artificial multicapa tradicional que usa una función de activación softmax para realizar la clasificación en la salida. El término ”Totalmente Conectado” se refiere a que cada neurona en la capa previa está conectada a cada neurona en la siguiente capa (Karn, 2016).

El flatten o aplanado es una operación que consiste en tomar ambos extremos de un volumen y ”Estirarlos” hasta conseguir un vector con todos los valores que contenía el volumen.

El resultado de aplicarle flatten al volumen de salida de la etapa de extracción de características (Convolución + ReLU + Pooling), es un vector que posee todas las características extraídas de la imagen de entrada, este vector servirá de entrada para esta capa, donde cada posición hará la función de una ”neurona” totalmente conectada a capas subyacentes (véase la Figura 4.9).

El objetivo de esta capa es ubicar a la imagen de entrada en una de las clases a detectar por la red, por ello la suma de las probabilidades de salida de la capa totalmente conectada debe ser uno. Esto se garantiza mediante el uso de Softmax como función de activación. La Figura 4.9 muestra la etapa de clasificación descrita.

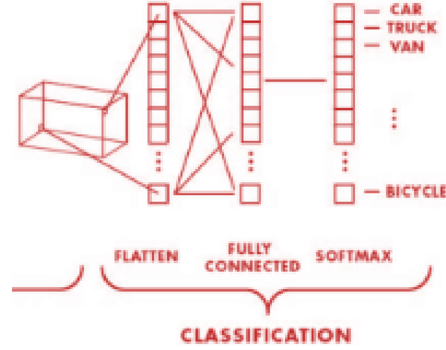

Figura 4.9: Etapa de clasificación CNNs
Adaptada de (Shyamal Patel, 2017)

Backpropagation

El algoritmo de backpropagation, o algoritmo de propagación hacia atrás, es el algoritmo por excelencia para entrenar con aprendizaje supervisado (dado un conjunto de datos etiquetados de entrenamiento) una red neuronal. Este algoritmo, al igual que muchos otros de aprendizaje supervisado, es un algoritmo de gradiente descendiente. Para cada entrada se calcula su salida como normalmente se haría, desde la capa de entrada hasta la de salida, y en la de salida se calcula la función de error (que suele ser el error cuadrático). A partir de ella la corrección del error se va propagando a las capas interiores, hacia la capa de input, de ahí su nombre, ya que va "hacia atrás" en relación a cómo la red funciona normalmente. La corrección del error se hace en cada neurona viendo el gradiente de la función de error y moviendo los parámetros en la dirección hacia donde éste desciende.

4.1.2 Entrenamiento

Como se mencionó anteriormente, las capas Convolución + ReLU + Pooling actúan como extractores de características de la imagen de entrada mientras que la capa totalmente conectada actúa como un clasificador.

El proceso general de entrenamiento de la red neuronal convolucional se puede resumir de la siguiente manera (Karn, 2016):

1. Elegir los hiperparámetros de la red.
2. Inicializar todos los parámetros (o pesos).
3. la red toma una imagen para entrenamiento como entrada, la imagen pasa por las etapas de extracción de características y clasificación. Al terminar el paso de la imagen por toda la arquitectura de la red, se cuenta con la probabilidad de que esta pertenezca a cada clase. Es importante destacar que como los pesos se asignan aleatoriamente para el primer ejemplo de entrenamiento, las probabilidades de salida también son aleatorias.

4. Calcular el error total en la capa de salida, con la siguiente fórmula: $ErrorTotal = \sum \frac{1}{2}(probObjetivo - probSalida)^2$ Donde la probabilidad objetivo viene dada por la etiqueta de la imagen y la probabilidad de salida es la obtenida por el paso de la imagen en la red.
5. Aplicar Backpropagation para calcular los gradientes del error con respecto a todos los pesos en la red. Usar el descenso de gradiente para actualizar todos los pesos de los filtros y demás parámetros de la red para minimizar el error de la salida. Los pesos se ajustan en proporción a su contribución al error total.
6. Repetir los pasos 2 a 4 con todas las épocas a entrenar.

4.1.3 Inferencia

El proceso de inferencia consiste en ingresar una nueva imagen (nunca antes vista) a la red neuronal convolucional ya entrenada, y observar si el proceso de clasificación es correcto. La imagen atraviesa toda la red (propagación hacia adelante) y genera una probabilidad para cada clase, esta probabilidad es calculada usando los pesos que se han aprendido para clasificar correctamente todos los ejemplos de entrenamiento. Si el conjunto de entrenamiento es lo suficientemente grande y representativo, la red clasificará bien las nuevas imágenes.

4.2 TensorFlow como framework de desarrollo

Aunque TensorFlow es popularmente usado en el ámbito del machine learning como framework, en general, es una librería para computación numérica, de código abierto y de alto rendimiento, de hecho, TensorFlow se usa para todo tipo de programación en GPU. Por ejemplo, se puede usar para resolver ecuaciones diferenciales parciales, las cuales son útiles en la dinámica de fluidos.

Además, permite escribir código en lenguajes de alto nivel como python y ejecutarlos de forma rápida, en consecuencia, su facilidad de uso, lo hace muy atractivo.

4.2.1 Tensor

Los tensores, son el tipo de datos que maneja TensorFlow; un tensor se define como un arreglo N-dimensional de datos, por ejemplo: un escalar es un tensor de dimensión cero, un vector es un tensor de dimensión uno y así sucesivamente. En la Figura 4.10 se muestra un ejemplo detallado de los tensores y su forma.

	Common name	Rank (Dimension)	Example	Shape of example
	Scalar	0	`x = tf.constant(3)`	()
	Vector	1	`x = tf.constant([3, 5, 7])`	(3,)
	Matrix	2	`x = tf.constant([[3, 5, 7], [4, 6, 8]])`	(2, 3)
	3D Tensor	3	`tf.constant([[[3, 5, 7],[4, 6, 8]], [[1, 2, 3],[4, 5, 6]] ])`	(2, 2, 3)

	nD Tensor	n	`x1 = tf.constant([2, 3, 4])`
			`x2 = tf.stack([x1, x1])`
			`x3 = tf.stack([x2, x2, x2, x2])`
			`x4 = tf.stack([x3, x3])`
				(3,)
				(2, 3)
				(4, 2, 3)
				(2, 4, 2, 3)

Figura 4.10: Tensores
Tomada de (Cloud, 2017)

4.2.2 Jerarquía de las capas de abstracción de TensorFlow

TensorFlow cuenta con cinco capas de abstracción diferentes (véase la Figura 4.11), cada una de ellas destinada a cumplir diversas funciones.

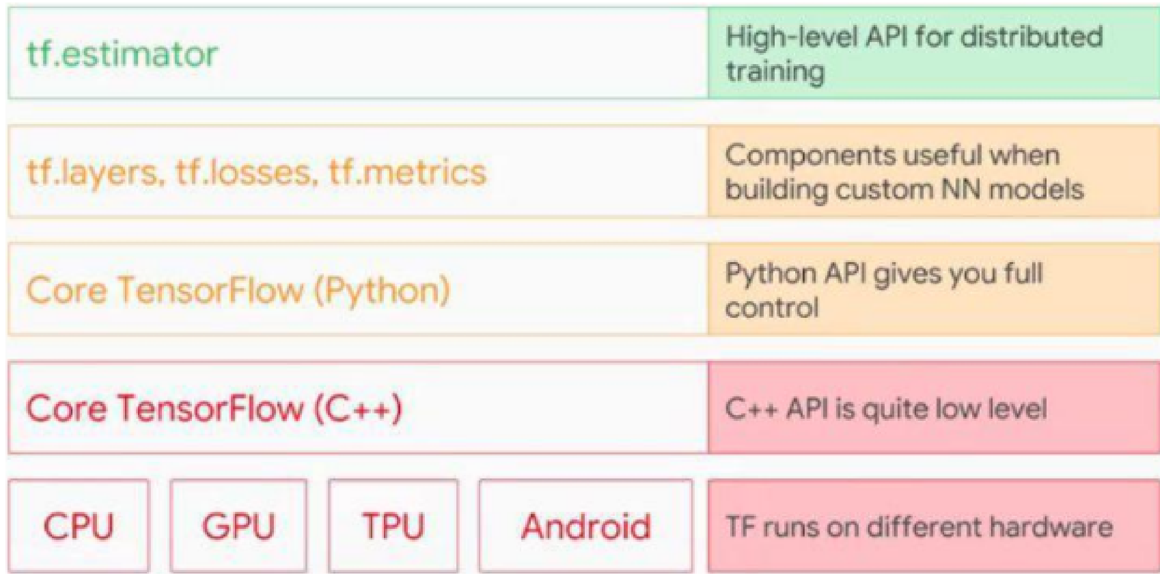

Figura 4.11: Jerarquía en TensorFlow
Tomada de (Cloud, 2017)

1. La primera capa y la más profunda está implementada con el fin de ejecutar tensorflow en diferentes plataformas de hardware, por ejemplo: CPUs, GPUs, TPUs, Android, IOS. . .

2. La segunda capa es el núcleo de TensorFlow, escrito en C++, permite añadir nuevos módulos o aplicaciones personalizadas a la librería. Contiene todas la operaciones básicas de TensorFlow, la cuales serán llamadas por la siguiente capa.

3. La tercera capa le pertenece al núcleo de TensorFlow en python, contiene gran parte del código de procesamiento numérico (sumas, restas, multiplicaciones...). Esta capa abstrae todas las funciones escritas en C++ y permite su acceso desde Python.

4. Esta capa contiene las definiciones de los módulos más usados para la construcción de modelos. Se encarga de hacer los llamados correspondientes a las funciones definidas en la capa anterior y los une para crear nuevos módulos cada vez más completos, los cuales están formados por clases que podemos llamar

fácilmente y solo pasarles los parámetros necesarios para su funcionamiento. Por ejemplo, el módulo tf.layers está compuesto por algunas clases como: la class AveragePooling1D (encargada de aplicar un pooling promedio en tensores 1D), class Conv2D(para convoluciones sobre tensores 2D), entre otros.

5. Muchos de los algoritmos usados en machine learning son repetitivos, por lo tanto, la capa estimator (la última y de más alto nivel), contiene modelos predefinidos para la definición, entrenamiento y pruebas de diferentes arquitecturas de machine learning. Además, estimator provee soluciones a cuatro de los problemas más grandes del mundo del aprendizaje automático como son (Cloud, 2017):

    - Creación de puntos de control o checkpoints del modelo: almacenar el estado actual del entrenamiento es muy conveniente al momento de entrenar modelos grandes, brindan continuidad en el entrenamiento, siendo útiles en el caso de una falla ya que permiten reanudar el entrenamiento desde cualquier checkpoint almacenado, además, almacena los modelos totalmente entrenados, de los cuales podemos hacer predicciones sin necesidad de volver a hacer el entrenamiento.
    - Manejo de grandes conjuntos de datos: tensorflow nos provee funciones que brindan la posibilidad de cargar progresivamente los datos a memoria principal, la idea principal es fragmentar un gran dataset en pequeños mini batches de datos para hacer el entrenamiento.
    - Entrenamiento distribuido: los modelos muy grandes pueden tomar horas, incluso días en entrenarse, la idea de distribuir el cómputo empieza a tomar valor en estos casos. Tensorflow proporciona las funciones necesarias para el manejo del paralelismo del entrenamiento de forma automática, la idea de su funcionamiento es la siguiente:
        (a) Se replica el modelo y se proporciona una copia a cada uno de los llamados trabajadores.
        (b) Se establecen los servidores centrales encargados de mantener el modelo completo entrenado, actualizando los pesos constantemente según sean proveídos por los trabajadores.

(c) En cada época (paso del conjunto de datos por el modelo) de entrenamiento, los trabajadores cargan un lote o batch de datos (con suerte diferente para cada uno) y los computan, al finalizar, los pesos aprendidos son enviados a los servidores centrales.

(d) Los servidores centrales actualizan el modelo principal, para luego enviarlo a todos los trabajadores en la siguiente época.

(e) Los trabajadores actualizan su réplica del modelo y continúan con otro batch de datos.

Esta función también se encarga del manejo de ckeckpoints en el sistema. La Figura 4.12 ilustra lo antes explicado.

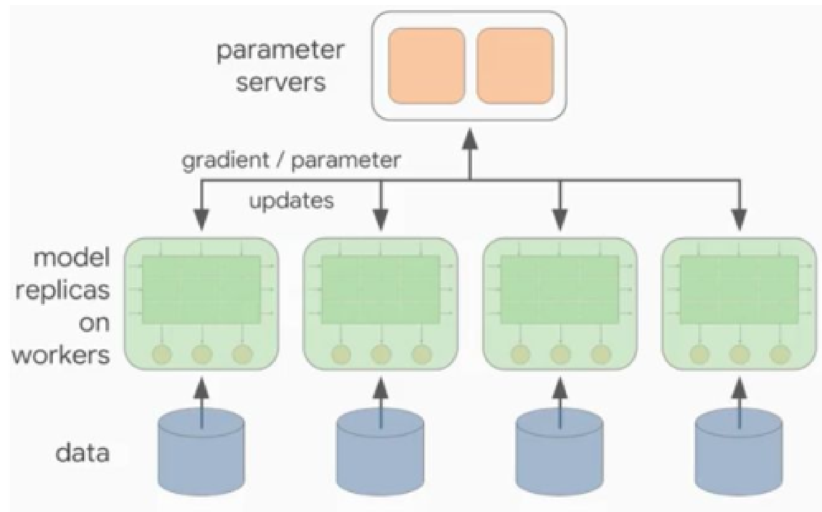

Figura 4.12: Entrenamiento distribuido
Tomada de (Cloud, 2017)

- Necesidad de evaluar el modelo durante el entrenamiento: TensorBoard (Figura 4.13) es una aplicación web que nos permite comparar y analizar el entrenamiento a medida que este ocurre, nos muestra gráficas de las métricas involucradas en el aprendizaje como también del grafo acíclico dirigido creado por tensorflow.

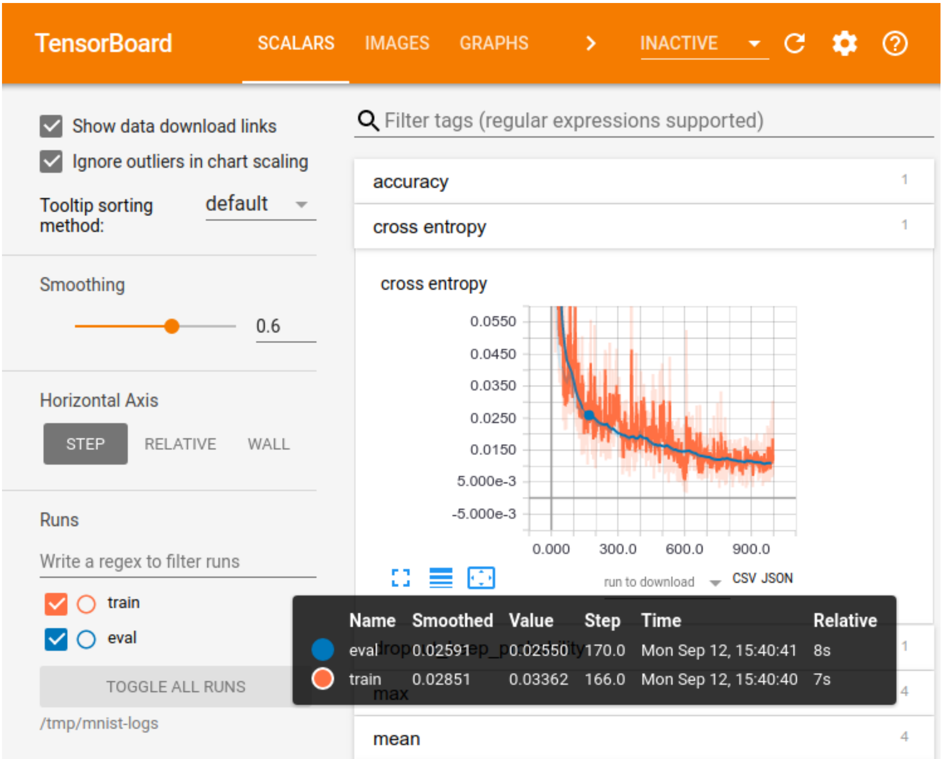

Figura 4.13: TensorBoard
Tomada de (TensorFlow, 2018c)

4.2.3 Representación de la operaciones en TensorFlow

A medida que se van declarando operaciones en el código, tensorflow va creando un grafo dirigido acíclico (DAG), en el cual los nodos representan las operaciones y los arcos el flujo de los datos en forma de tensores (Cloud, 2017), por ejemplo, si se desea hacer una simple suma usando tensorflow, el código sería el siguiente:

$c = tf.add(a, b)$ donde a,b y c son tensores y add es la operación de suma

La Figura 4.14 muestra el DAG creado por tensorflow para esta operación

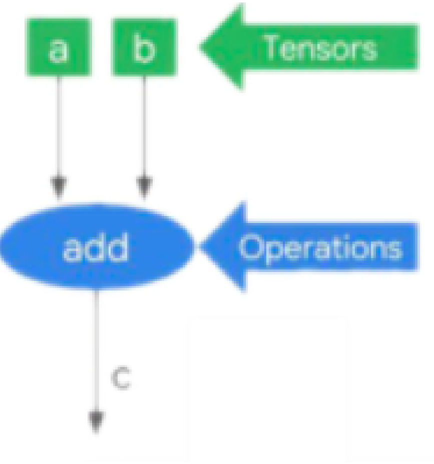

Figura 4.14: Ejemplo de DAG
Tomada de (Cloud, 2017)

Formas de ejecución del DAG

1. Lazy Evaluation (evaluación perezosa): En producción es la forma de evaluación más usada, cuenta con dos etapas, la primera es la etapa de construcción, donde se crea el DAG y la segunda es la etapa de ejecución, donde se ejecuta el DAG. Por lo tanto, no hay evaluación inmediata, hasta que se especifique explícitamente en el código. Algunas de las ventajas que nos trae este modo de evaluación son las siguientes:
    - Se pueden asignar diferentes partes del grafo a diferentes dispositivos para su ejecución, por lo tanto es muy fácil paralelizar el código.
    - A medida que se crea el grafo el sistema de ejecución de tensorflow lo va optimizando, además, se encarga automáticamente de distribuir el grafo.
2. Eager Evaluation (evaluación ansiosa): al contrario de la lazy evaluation en este modo de ejecución el grafo se va ejecutando de forma inmediata. Es poco usada, su principal función es servir de apoyo al programador cuando desea depurar el código.

4.2.4 Perfilado temporal en TensorFlow

Tensorflow nos provee un API llamada tf.profiler.Profiler especialmente diseñada para aplicar el perfilado temporal al DAG. Nos permite observar el tiempo de ejecución total en CPU y el uso en memoria de cada una de las operaciones declaradas en el grafo, para esto hace uso de una interfaz grafica llamada Profiler (véase la Figura 4.15) o del terminal (véase la Figura 4.16).

Figura 4.15: Interfaz gráfica Profiler

```
==================Model Analysis Report======================

Doc:
op: The nodes are operation kernel type, such as MatMul, Conv2D. Graph nodes belonging to the same type are aggregated together.
requested bytes: The memory requested by the operation, accumulatively.
total execution time: Sum of accelerator execution time and cpu execution time.
cpu execution time: The time from the start to the end of the operation. It's the sum of actual cpu run time plus the time that it spends waiting if part of computation is launched a
synchronously.
accelerator execution time: Time spent executing on the accelerator. This is normally measured by the actual hardware library.

Profile:
node name | requested bytes | total execution time | accelerator execution time | cpu execution time
Conv2D                     22.84MB (100.00%, 31.86%),      32.51ms (100.00%, 74.82%),            0us (0.00%, 0.00%),      32.51ms (100.00%, 74.82%)
LRN                          7.93MB (68.14%, 11.06%),         3.73ms (25.18%, 8.57%),            0us (0.00%, 0.00%),         3.73ms (25.18%, 8.57%)
BiasAdd                           0B (0.00%, 0.00%),         2.15ms (16.61%, 4.95%),            0us (0.00%, 0.00%),         2.15ms (16.61%, 4.95%)
MaxPool                       5.14MB (57.08%, 7.18%),         1.99ms (11.65%, 4.59%),            0us (0.00%, 0.00%),         1.99ms (11.65%, 4.59%)
Relu                              0B (0.00%, 0.00%),          1.52ms (7.07%, 3.50%),            0us (0.00%, 0.00%),          1.52ms (7.07%, 3.50%)
MatMul                     300.03KB (49.91%, 0.42%),          1.15ms (3.57%, 2.65%),            0us (0.00%, 0.00%),          1.15ms (3.57%, 2.65%)
QueueDequeueManyV2         885.25KB (49.49%, 1.23%),           244us (0.91%, 0.56%),            0us (0.00%, 0.00%),           244us (0.91%, 0.56%)
Add                               0B (0.00%, 0.00%),            81us (0.35%, 0.19%),            0us (0.00%, 0.00%),            81us (0.35%, 0.19%)
VariableV2                    4.27MB (48.25%, 5.96%),           30us (0.17%, 0.07%),            0us (0.00%, 0.00%),            30us (0.17%, 0.07%)
Const                           20B (42.29%, 0.00%),            12us (0.10%, 0.03%),            0us (0.00%, 0.00%),            12us (0.10%, 0.03%)
FIFOQueueV2                30.32MB (42.29%, 42.29%),            11us (0.07%, 0.03%),            0us (0.00%, 0.00%),            11us (0.07%, 0.03%)
Identity                          0B (0.00%, 0.00%),            10us (0.04%, 0.02%),            0us (0.00%, 0.00%),            10us (0.04%, 0.02%)
Reshape                           0B (0.00%, 0.00%),             5us (0.02%, 0.01%),            0us (0.00%, 0.00%),             5us (0.02%, 0.01%)
InTopKV2                        128B (0.00%, 0.00%),             4us (0.01%, 0.01%),            0us (0.00%, 0.00%),             4us (0.01%, 0.01%)

======================End of Report==========================
2018-10-04 14:28:26.522944: precision @ 1 = 0.863
```

Figura 4.16: Perfilado temporal, resultados del terminal

4.2.5 Ventajas y desventajas de TensorFlow

Ventajas

- TensorFlow es el framework más usado tanto a nivel académico (20.800 resultados en google scholar) como a nivel de producción para machine learning y es de código abierto, por lo tanto, posee una gran cantidad de funciones ya hechas, tanto las clásicas como nuevas implementaciones, basadas en el estado del arte, incluso, en su documentación, muchas de estas funciones tienen enlaces directos a los papers donde sus creadores explican los hallazgos que obtuvieron con ellas.
- Con el uso de estimator, se pueden crear modelos completos sumamente fácil.
- TensorFlow se encarga automáticamente de paralelizar el código.
- Se puede ajustar a un API llamada Keras, es de más alto nivel comparada con estimator, por lo tanto, prácticamente cualquier persona puede usar TensorFlow.

Desventajas

- La versión 1.10 de TensorFlow no sirve en muchos cpu y no hay modo de saber esto antes de instalarlo. Esto sucede porque la versión 1.10 utiliza un set de instrucciones que no está disponible en muchos cpus. La solución es compilarla directamente sin incluir esas instrucciones pero la documentación para esto es totalmente deficiente.

- Mientras más de alto nivel es la función, mayor documentación tiene, de este modo, si una persona sólo desea crear modelos, entrenarlos y probarlos, va a ser realmente fácil el uso de TensorFlow. Por lo contrario, las funciones de niveles más bajos en la jerarquía carecen en muchos casos de documentaciones explícitas, sin embargo, como la comunidad de TensorFlow es tan grande, poco a poco se logran encontrar respuestas.
- La función tf.profile ralentiza el proceso de entrenamiento unas cinco veces, además su documentación es realmente pobre.

4.3 Perfilado y optimización

4.3.1 Dataset de entrenamiento CIFAR-10

CIFAR-10 es un dataset de referencia común en el aprendizaje automático. Contiene 60.000 imágenes a color de tamaño 32x32 cada una con su etiqueta, divididas en 10 clases (avión, carro, ave, gato, venado, perro, rana, caballo, barco y camión), 50.000 imágenes son para el proceso de entrenamiento y 10.000 para el proceso de inferencia. Es altamente utilizado en competencias y en el ámbito investigativo como dataset de prueba. Fue recolectado por Alex Krizhevsky, Vinod Nair, y Geoffrey Hinton.

4.3.2 Arquitectura e implementación del modelo

El modelo aquí expuesto, sigue la arquitectura descrita por (Krizhevsky, 2011), con algunas diferencias en las capas superiores. Obtiene una precisión de aproximadamente 86%.

El código base para los experimentos realizados, fue obtenido de (Authors, 2018)

Organización del código

Tabla 4.1: Descripción de los módulos del código

Archivo	Descripción
cifar10_input.py	Lee el dataset CIFAR-10 en formato binario y se encarga de construir entradas adecuadas para alimentar al modelos de datos.
cifar10.py	Construye la arquitectura de la red neuronal convolucional, contiene la definición de la función de inferencia de la red (función de interés en este estudio).
cifar10_train.py	Entrena la red con el modelo provisto por cifar10.py
cifar10_eval.py	Evalúa y predice el rendimiento del modelo.

Arquitectura de la función de inferencia

Como fue mencionado anteriormente, la parte del código encargada de construir la arquitectura de la red neuronal y hacer la predicción sobre el modelo, está programada en la función inference() dentro de cifar10.py. Esta función está organizada como se ve en la Tabla 4.2.

Cabe destacar que en este modelo se deben aprender alrededor de 1.068.298 parámetros y se requieren hacer 19.5M de multiplicaciones y adiciones para computar la inferencia en una sola imagen (TensorFlow, 2018a).

La Figura 4.17 muestra el DAG creado por tensorflow para esta función.

Tabla 4.2: Descripción de la función inference()

Capa	Descripción
conv1	Esta capa usa tf.nn.conv2d para hacer la convolución y tf.nn.relu como función de activación
pool1	Aplicación de pooling con tf.nn.max_pool.
norm1	La salida de la capa pool1 se normaliza con tf.nn.local_response_normalization, La normalización es útil para evitar que las neuronas se saturen cuando las entradas pueden tener una escala variable (TensorFlow, 2018d).
conv2	Esta capa usa tf.nn.conv2d para hacer la convolución y tf.nn.relu como función de activación.
norm2	La salida de la capa conv2 se normaliza con tf.nn.local_response_normalization.
pool2	Aplicación de pooling con tf.nn.max_pool.
local3	Capa totalmente conectada con función de activación lineal rectificada (TensorFlow, 2018b).
local4	Capa totalmente conectada con función de activación lineal rectificada
softmax_linear	Produce la distribución de probabilidad para cada clase

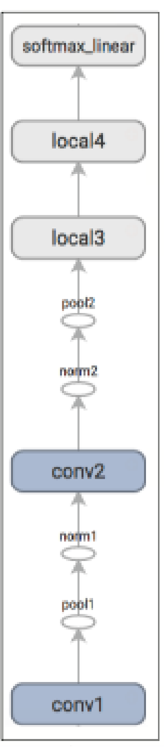

Figura 4.17: Arquitectura de la función de inferencia
Tomada de (Cloud, 2017)

Para la realización de los experimentos, la función a modificar es inference(), esta función está implementada en el código original de TensorFlow (Authors, 2018) de la siguiente forma:

```
def inference(images):
  # conv1
  with tf.variable_scope('conv1') as scope:
    kernel = _variable_with_weight_decay('weights',
                                         shape=[5, 5, 3, 64],
                                         stddev=5e-2,
                                         wd=None)
    conv = tf.nn.conv2d(images, kernel, [1, 1, 1, 1],
                        padding='SAME')
    biases = _variable_on_cpu('biases', [64],
                              tf.constant_initializer(0.0))
    pre_activation = tf.nn.bias_add(conv, biases)
    conv1 = tf.nn.relu(pre_activation, name=scope.name)
    _activation_summary(conv1)
  # pool1
  pool1 = tf.nn.max_pool(conv1, ksize=[1, 3, 3, 1],
                         strides=[1, 2, 2, 1],
                         padding='SAME', name='pool1')
  # norm1
  norm1 = tf.nn.lrn(pool1, 4, bias=1.0, alpha=0.001 / 9.0,
                    beta=0.75, name='norm1')

  # conv2
  with tf.variable_scope('conv2') as scope:
    kernel = _variable_with_weight_decay('weights',
                                         shape=[5, 5, 64, 64],
                                         stddev=5e-2,
                                         wd=None)
    conv = tf.nn.conv2d(norm1, kernel, [1, 1, 1, 1],
                        padding='SAME')
```

```
    biases = _variable_on_cpu('biases', [64],
                              tf.constant_initializer(0.1))
    pre_activation = tf.nn.bias_add(conv, biases)
    conv2 = tf.nn.relu(pre_activation, name=scope.name)
    _activation_summary(conv2)

  # norm2
  norm2 = tf.nn.lrn(conv2, 4, bias=1.0, alpha=0.001 / 9.0,
                    beta=0.75, name='norm2')

  # pool2
  pool2 = tf.nn.max_pool(norm2, ksize=[1, 3, 3, 1],
                         strides=[1, 2, 2, 1], padding='SAME',
                         name='pool2')
  # local3
  with tf.variable_scope('local3') as scope:
    reshape = tf.reshape(pool2,
                         [images.get_shape().as_list()[0], -1])
    dim = reshape.get_shape()[1].value
    weights = _variable_with_weight_decay('weights',
                                          shape=[dim, 384],
                                          stddev=0.04,
                                          wd=0.004)
    biases = _variable_on_cpu('biases', [384],
                              tf.constant_initializer(0.1))
    local3 = tf.nn.relu(tf.matmul(reshape, weights) + biases,
                        name=scope.name)
    _activation_summary(local3)

  # local4
  with tf.variable_scope('local4') as scope:
    weights = _variable_with_weight_decay('weights',
                                          shape=[384, 192],
```

```
                                          stddev=0.04,
                                          wd=0.004)
    biases = _variable_on_cpu('biases', [192],
                              tf.constant_initializer(0.1))
    local4 = tf.nn.relu(tf.matmul(local3, weights) + biases,
                        name=scope.name)
    _activation_summary(local4)


  #softmax
  with tf.variable_scope('softmax_linear') as scope:
    weights = _variable_with_weight_decay('weights',
                                          [192, NUM_CLASSES],
                                          stddev=1/192.0,
                                          wd=None)
    biases = _variable_on_cpu('biases', [NUM_CLASSES],
                               tf.constant_initializer(0.0))
    softmax_linear = tf.add(tf.matmul(local4, weights), biases,
                            name=scope.name)
    _activation_summary(softmax_linear)

  return softmax_linear
```

4.4 Análisis de los cuellos de botella: Perfilado temporal

Para medir el tiempo original de ejecución en CPU y el uso de memoria de las operaciones involucradas en la función inferece(), se hizo el perfilado temporal a cien épocas del módulo cifar10_eval.py, los resultados obtenidos sobre estas operaciones se enseñan en la Figura 4.18 y en la Tabla 4.3:

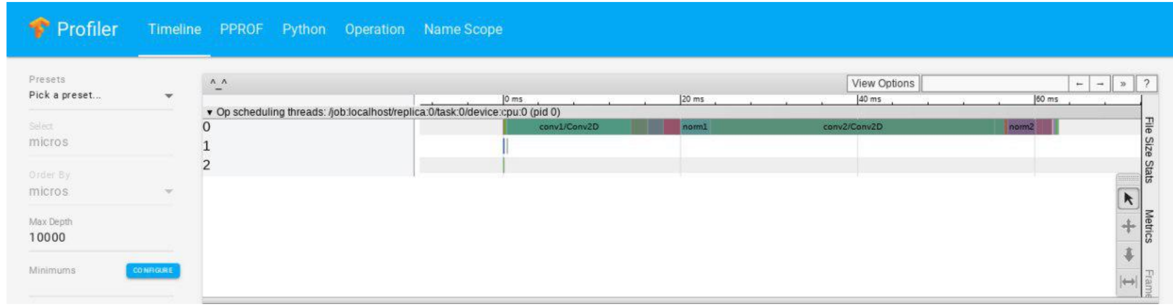

Figura 4.18: Resultados perfilado temporal inference()

Tabla 4.3: Resultados promedio perfilado temporal inferece()

Operación	Tiempo de ejecución en CPU	Bytes usados
Conv2D	40.34ms (100.00%, 72.02%)	25.55MB (100.00%, 28.08%)
LRN	5.78ms (27.98%, 10.33%)	9.44MB (24.32%, 10.37%)
BiasAdd	2.67ms (17.65%, 4.76%)	0B (0.00%, 0.00%)
MaxPool	2.38ms (12.89%, 4.26%)	6.82MB (13.52%, 7.49%)
Relu	2.22ms (8.63%, 3.97%)	0B (0.00%, 0.00%)
MatMul	1.57ms (4.66%, 2.81%)	385.00KB (13.95%, 0.42%)
QueueManyV2	687us (1.85%, 1.23%)	0B (0.00%, 0.00%)
Add	136us (0.62%, 0.24%)	0B (0.00%, 0.00%)
InTopKV2	115us (0.38%, 0.21%)	256B (24.32%, 0.00%)
VariableV2	49us (0.18%, 0.09%)	4.52MB (4.97%, 4.97%)
Const	23us (0.09%, 0.04%)	1.02KB (100.00%, 0.00%)
FIFOQueueV2	12us (0.05%, 0.02%)	43.30MB (71.92%, 47.60%)
Identity	10us (0.03%, 0.02%)	0B (0.00%, 0.00%)
Reshape	4us (0.01%, 0.01%)	0B (0.00%, 0.00%)

Podemos observar claramente que la operación más costosa computacionalmente es la convolución, según los resultados del perfilado temporal, usa un 72% del 100% total del tiempo de ejecución en CPU, seguido de la normalización que usa un 10.33%. En cuanto al uso de memoria la operación más costosa es FIFOQueueV2 con un 47.60%, seguida de la convolución con 28.08%. Es de esperar, que si se desea optimizar este código, se debe prestar especial atención en mejorar la operación de convolución.

4.5 Propuestas para la mejora del rendimiento

Como se pudo observar en el punto anterior, el cuello de botella más grande en el proceso de inferencia en una red neuronal convolucional se encuentra en la capa convolucional, esto se debe a la gran cantidad de multiplicaciones que debe hacer por paso de cada imagen, para la mejora de este código se realizaron las siguientes propuestas:

1. Disminuir la cantidad de multiplicaciones que debe hacer la capa convolucional, usando un filtro con mayor stride, ya que el stride del código original es de 1 (como se puede observar en el código expuesto en la sección 4.4.2), los experimentos se realizaron modificando el código original de ambas capas convolucionales, en el primero se aumento el stride a 2 (véase la Tabla 4.4), en el segundo se aumento el stride a 3 (véase la Tabla 4.5) y en el tercero se aumento el stride a 5 (véase la Tabla 4.6), obteniendo los siguientes resultados promedios al ejecutar 100 epocas en cifar10_eval.py:

Tabla 4.4: Stride 2: Perfilado temporal en inference(), precisión: 79.6%

Operación	**Tiempo de ejecución en CPU**	**Bytes usados**
Conv2D	6.54ms (100.00%, 64.67%)	5.01MB (100.00%, 12.40%)
LRN	1.15ms (35.33%, 11.37%)	1.47MB (87.60%, 3.65%)
BiasAdd	605us (23.97%, 5.98%)	1.31MB (83.95%, 3.24%)
MaxPool	548us (17.98%, 5.42%)	0B (0.00%, 0.00%)
Relu	458us (12.56%, 4.53%)	300.03KB (80.71%, 0.74%)
MatMul	364us (8.03%, 3.60%)	0B (0.00%, 0.00%)
QueueManyV2	242us (4.43%, 2.39%)	885.25KB (79.97%, 2.19%)
Add	134us (2.04%, 1.33%)	0B (0.00%, 0.00%)
InTopKV2	32us (0.71%, 0.32%)	1.13MB (77.78%, 2.79%)
VariableV2	32us (0.71%, 0.32%)	30.32MB (74.99%, 74.99%)
Const	14us (0.40%, 0.14%)	20B (0.00%, 0.00%)
FIFOQueueV2	12us (0.26%, 0.12%)	0B (0.00%, 0.00%)
Identity	10us (0.14%, 0.10%)	0B (0.00%, 0.00%)

Tabla 4.5: Stride 3: Perfilado temporal en inference(), precisión: 74.7%

Operación	Tiempo de ejecución en CPU	Bytes usados
Conv2D	2.61ms (100.00%, 56.73%)	2.26MB (100.00%, 4.74%)
LRN	531us (43.27%, 11.52%)	698.83KB (95.26%, 1.46%)
BiasAdd	333us (31.75%, 7.23%)	885.29KB (93.80%, 1.85%)
MaxPool	312us (24.52%, 6.77%)	373.96KB (91.95%, 0.78%)
Relu	249us (17.75%, 5.40%)	0B (0.00%, 0.00%)
MatMul	248us (12.35%, 5.38%)	735.88KB (91.16%, 1.54%)
QueueManyV2	116us (6.97%, 2.52%)	0B (0.00%, 0.00%)
Add	111us (4.45%, 2.41%)	0B (0.00%, 0.00%)
InTopKV2	45us (2.04%, 0.98%)	1.04MB (89.62%, 2.18%)
VariableV2	19us (1.06%, 0.41%)	1.02KB (87.44%, 0.00%)
Const	13us (0.65%, 0.28%)	41.74MB (87.44%, 87.44%)
FIFOQueueV2	10us (0.37%, 0.22%)	0B (0.00%, 0.00%)
Identity	7us (0.15%, 0.15%)	256B (0.00%, 0.00%)

Tabla 4.6: Stride 5: Perfilado temporal en inference(), precisión: 70%

Operación	Tiempo de ejecución en CPU	Bytes usados
Conv2D	1.24ms (100.00%, 43.69%)	862.61KB (100.00%, 1.86%)
LRN	341us (56.31%, 12.02%)	885.28KB (98.14%, 1.91%)
BiasAdd	327us (44.29%, 11.52%)	367.25KB (96.23%, 0.79%)
MaxPool	321us (32.77%, 11.31%)	485.14KB (95.43%, 1.05%)
Relu	159us (21.46%, 5.60%)	419.73KB (94.39%, 0.91%)
MatMul	136us (15.86%, 4.79%)	0B (0.00%, 0.00%)
QueueManyV2	125us (11.06%, 4.40%)	0B (0.00%, 0.00%)
Add	99us (6.66%, 3.49%)	0B (0.00%, 0.00%)
InTopKV2	44us (3.17%, 1.55%)	832.77KB (93.48%, 1.80%)
VariableV2	18us (1.62%, 0.63%)	1.02KB (91.68%, 0.00%)
Const	13us (0.99%, 0.46%)	42.46MB (91.68%, 91.68%)
FIFOQueueV2	10us (0.53%, 0.35%)	0B (0.00%, 0.00%)
Identity	5us (0.18%, 0.18%)	256B (0.00%, 0.00%)

Los resultados obtenidos en estos experimentos son realmente sorprendentes, aunque la convolución sigue siendo la operación más costosa, podemos notar como aumentar el stride de los filtros disminuye indudablemente el tiempo de ejecución total por paso de cada imagen en la red, de 60ms aprox. en el proceso original a 9ms aprox. con un stride de 2, luego a 4ms aprox. con un stride de 3 y por último a 2ms aprox. con un stride de 5, todo esto, sin dismunir excesivamente la precisión de la inferencia.

Si observamos el tiempo promedio de ejecución de conv2D con respecto a todas las operaciones, notamos que este disminuye progresivamente de un experimento a otro, en la tabla 4.3 en promedio el tiempo de ejecución es de 72.02%, con stride 2 es de 64.67%, con stride 3 es de 56.73% y con stride 5 es de 43.69%, por lo que es evidente que el aumento en el stride, es una mejora en cuanto a tiempo de ejecución en CPU de las capas convolucionales.

Con respecto al uso de memoria total por la operación de convolución en la red, también podemos notar mejoras valiosas, de 25.55MB en el resultado original a 862.61KB aplicando un stride de 5, practicamente se reduce un 100% el uso de memoria de esta operación.

Cabe destacar, aunque no es este nuestro caso de estudio, que estas mejoras también se observaron en el proceso de entrenamiento de la red neuronal convolucional, la red original demoró alrededor de 8 horas en entrenarse con 100.000 epocas a sólo 30 min con stride 5 en la misma cantidad de épocas.

2. Disminuir la cantidad de multiplicaciones que debe hacer la capa convolucional, reduciendo la cantidad de filtros a la mitad (de 64 a 32), los resultados se muestran en la Tabla 4.7:

Tabla 4.7: Reducción de la cantidad de filtros: Perfilado temporal en inference(), precisión: 83.7%

Operación	Tiempo de ejecución en CPU	Bytes usados
Conv2D	20.97ms (100.00%, 75.14%)	11.80MB (100.00%, 22.19%)
LRN	2.27ms (24.86%, 8.12%)	4.72MB (77.81%, 8.88%)
BiasAdd	1.48ms (16.74%, 5.30%)	2.95MB (68.93%, 5.55%)
MaxPool	1.18ms (11.44%, 4.22%)	0B (0.00%, 0.00%)
Relu	865us (7.21%, 3.10%)	0B (0.00%, 0.00%)
MatMul	740us (4.11%, 2.65%)	300.03KB (63.38%, 0.56%)
QueueManyV2	246us (1.46%, 0.88%)	885.25KB (62.82%, 1.67%)
Add	89us (0.58%, 0.32%)	0B (0.00%, 0.00%)
InTopKV2	30us (0.26%, 0.11%)	2.19MB (61.15%, 4.11%)
VariableV2	14us (0.15%, 0.05%)	30.32MB (57.04%, 57.04%)
Const	10us (0.10%, 0.04%)	20B (0.00%, 0.00%)
FIFOQueueV2	10us (0.07%, 0.04%)	0B (0.00%, 0.00%)
Identity	5us (0.03%, 0.02%)	0B (0.00%, 0.00%)

Reducir la cantidad de filtros a la mitad, logra reducir justamente a la mitad el tiempo de ejecución total en CPU y el uso en memoria en la red, reduciendo solamente un 3% la precisión de la CNN. Esto puede ser debido a que estaba ocurriendo un sobreajuste en la red, y en vez de mejorar su precisión con los 64 filtros llegó un punto en el que empezó a empeorar.

Con respecto al tiempo de ejecucion y uso en memoria porcentual de la capa convolucional, no se nota una mejora. Esto debido a que computacionalmente, esta capa sigue haciendo la misma cantidad de multiplicaciones pero para menos filtros, no se está alterando el código como tal de la convolución.

3. Disminuir la cantidad de multiplicaciones que debe hacer la capa convolucional, aumentando el tamaño del pooling en la primera capa convolucional para reducir el tamaño de la entrada de la segunda capa convolucional.

Tabla 4.8: Aumento de pooling: Perfilado temporal en inference(), precisión: 86.3%

Operación	Tiempo de ejecución en CPU	Bytes usados
Conv2D	32.51ms (100.00%, 74.82%)	22.84MB (100.00%, 31.86%)
LRN	3.73ms (25.18%, 8.57%)	7.93MB (68.14%, 11.06%)
BiasAdd	2.15ms (16.61%, 4.95%)	0B (0.00%, 0.00%)
MaxPool	1.99ms (11.65%, 4.59%)	5.14MB (57.08%, 7.18%)
Relu	1.52ms (7.07%, 3.50%)	0B (0.00%, 0.00%)
MatMul	1.15ms (3.57%, 2.65%)	300.03KB (49.91%, 0.42%)
QueueManyV2	244us (0.91%, 0.56%)	885.25KB (49.49%, 1.23%)
Add	81us (0.35%, 0.19%)	0B (0.00%, 0.00%)
InTopKV2	30us (0.17%, 0.07%)	4.27MB (48.25%, 5.96%)
VariableV2	12us (0.10%, 0.03%)	20B (42.29%, 0.00%)
Const	11us (0.07%, 0.03%)	30.32MB (42.29%, 42.29%)
FIFOQueueV2	10us (0.04%, 0.02%)	0B (0.00%, 0.00%)
Identity	5us (0.02%, 0.01%)	0B (0.00%, 0.00%)

Este esperimento fue propuesto, en vista de que parecia ser una buena idea reducir el tamaño de los datos de una capa convolucional a la otra para conseguir mejores resultados pero si comparamos la Tabla 4.3 con la Tabla 4.8 podemos observar que no hay una mejora notable en los resultados.

4.6 Conocimientos adquiridos durante la carrera vinculados al desarrollo del proyecto

Como estudiantes de ingeniería en sistemas, desarrollamos un background muy rico de habilidades, lo que nos permite desenvolvernos de manera exitosa en cualquier actividad que deseemos realizar, es un poco difícil para mí asociar los conocimientos específicos que necesité en el transcurso de mi tiempo como pasante en MeridaTech, pues, considero que la mezcla de todos ellos me ayudaron a culminarlas con buenos resultados.

Sin embargo, si debo hacer hincapié en algunas materias, seguro los conocimientos adquiridos en materias como inteligencia artificial, estocástica, matemáticas especiales

e introducción al control automático me fueron de gran utilidad al consultar conceptos más profundos sobre el mundo del machine learning.

Capítulo 5

Conclusiones

Con respecto a los experimentos realizados, aunque los mejores resultados se obtuvieron al aumentar el tamaño de stride, es importante tener cuidado con la perdida de precisión de la red neuronal al momento de hacer predicciones que esto conlleva; por otra parte, la reducción de la cantidad de filtros tiene resultados aceptables en cuanto precisión, pero no mejora mucho el tiempo de ejecución en CPU de las convoluciones, futuros experimentos podrían ser unir estas dos propuestas y ver que resultados se obtienen después de ello.

TensorFlow definitivamente es un framework muy prometedor en el mundo del machine learning, aunque muchas veces fue complicado trabajar con funciones de bajo nivel por falta de documentación, después de comprender su uso, logré cosas maravillosas, un ejemplo de esto, el análisis del perfilado temporal. Puedo notar el potencial que tiene y espero siga creciendo la comunidad de desarrolladores en TensorFlow, seguro poco a poco mejoran las deficiencias que tiene hasta ahora.

MeridaTech es sin duda alguna el mejor lugar que pude elegir para crecer profesional y personalmente, llevo muchas lecciones de vida y espero aplicarlas siempre, estoy enormemente agradecida por la oportunidad de trabajar con personas tan valiosas, y que no dudaban nunca en compartir sus conocimientos conmigo.

Bibliografía

Authors, T. (2018). Advanced Convolutional Neural Networks. https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10. Fecha de consulta:20 de Septiembre de 2018.

Cloud, G. (2017). Machine Learning with TensorFlow on Google Cloud Platform Specialization. https://www.coursera.org/learn/intro-tensorflow. Fecha de consulta:25 de Septiembre de 2018.

Dertat, A. (2017). Applied Deep Learning - Part 4: Convolutional Neural Networks. https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2. Fecha de consulta: 20 de Septiembre de 2018.

Fei Fei Li, Justin Johnson, S. Y. (2017). Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/convolutional-networks/. Fecha de consulta: 11 de Septiembre de 2018.

Fei Fei Li, Justin Johnson, S. Y. (2018). Lecture 5:Convolutional Neural Networks. chrome-extension://oemmndcbldboiebfnladdacbdfmadadm/http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture05.pdf. Fecha de consulta: 15 de Septiembre de 2018.

Karn, U. (2016). An Intuitive Explanation of Convolutional Neural Networks. https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/. Fecha de consulta:11 de Septiembre de 2018.

Krizhevsky, A. (2011). cuda-convnet. https://code.google.com/archive/p/cuda-convnet/. Fecha de consulta:19 de Septiembre de 2018.

MeridaTech (2018). Merida Technology Group C.A. (MeridaTech). http://www.meridatech.com/. Fecha de consulta: 18 de Septiembre de 2018.

Ng, A. (2018). Convolutional Neural Networks. https://www.coursera.org/learn/convolutional-neural-networks/home/welcome. Fecha de consulta: 1 de Septiembre de 2018.

Shyamal Patel, J. P. (2017). Introduction to Deep Learning: What Are Convolutional Neural Networks? https://www.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks--1489512765771.html. Fecha de consulta: 12 de Septiembre de 2018.

TensorFlow (2018a). Advanced Convolutional Neural Networks. https://www.tensorflow.org/tutorials/images/deep_cnn. Fecha de consulta:25 de Septiembre de 2018.

TensorFlow (2018b). Neural Network. https://www.tensorflow.org/api_guides/python/nn. Fecha de consulta:11 de Septiembre de 2018.

TensorFlow (2018c). TensorBoard: Visualizing Learning. https://www.tensorflow.org/guide/summaries_and_tensorboard. Fecha de consulta:1 de Octubre de 2018.

TensorFlow (2018d). tf.nn.local$_{r}esponse_{n}ormalization$. . Fecha de consulta:11 de Septiembre de 2018.