Deep Learning CNN in Tensorflow with GPU.

Bharath Anand[1]
Graduate
Department of Data Science
Professor Alfa Heryudono, University of Massachusetts Dartmouth

1. Abstract

Convolutional neural networks (CNNs), one of the most significant deep learning models, have excelled in a variety of applications including image classification, audio recognition, and interpreting natural language. Due to the high computational cost of training CNNs on big data sets, there has been a flurry of research and development of open-source GPU implementations. However, there aren't many research that have been done to assess the performance traits of their implementations. In this article, we thoroughly compare these implementations.

2. Introduction

Convolutional neural networks (CNNs) are significant deep learning models that have excelled at speech recognition, understanding natural language, and large-scale picture classification. This is explained by the sophisticated architecture of CNNs (such AlexNet, VGGNet, GoogleNet, and OverFeat), the size of the labeled training data, and the processing power of GPUs.

CNN training is incredibly expensive for two reasons. First, as depth and parameters are raised, CNNs are becoming more complex. It takes a long time to train these massive CNNs because it requires thousands of iterations of forward and backward propagations. Second, the size of the training samples is increasing. CNNs must be trained on some very huge datasets, driven by industry groups like Google, YouTube, Twitter, and Facebook (e.g., text, audio and video). Again, training on those huge datasets involves a significant amount of runtime; it's not unusual for this to be several weeks or months.

It is common practice to use GPUs to speed up CNN training to address this difficulty. The computation required for CNN training is massively parallel and heavily reliant on floating-point operations, such as matrix and vector computations. The GPU computing model works well with this computing pattern.

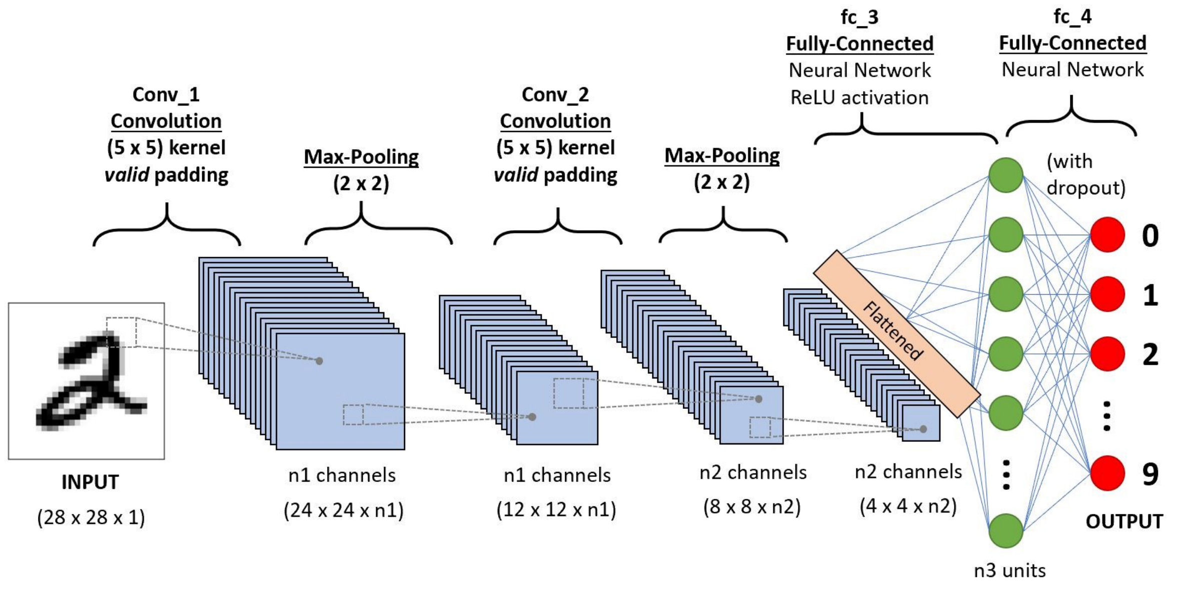


Fig 1. A simple CNN Architecture.

2. Background

Better comprehension of CNN architecture is essential for evaluating and optimizing convolution implementations. In this section, we give a general overview of CNN architecture and talk about several convolution techniques used in conventional CNN implementations.

2.1. Convolutional Neural Networks

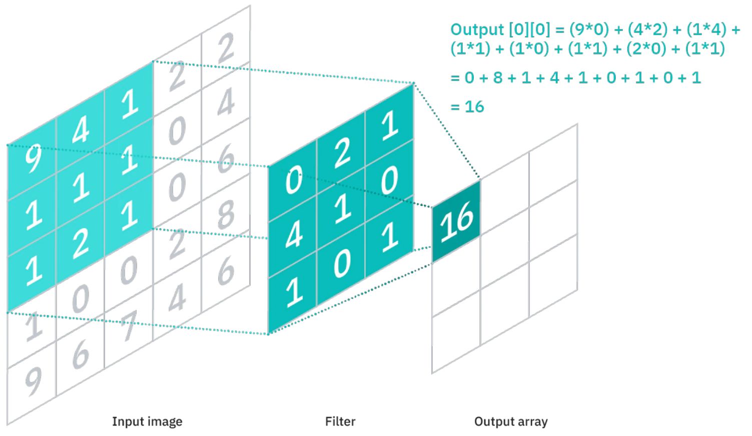
CNNs are trained using a standard feed-forward neural network, which employs the BP method to modify the learnable kernels in order to reduce the cost function. Three fundamental concepts local receptive field, shared weight, and pooling allow convolutional neural networks to automatically give some degree of shift and distortion invariance.

The core of CNNs is the convolutional layer. Each neuron of the same feature map applies the same weights over the input data at all feasible points in the convolutional layer in order to extract the appropriate features. The results of the convolving are arranged into a collection of two-dimensional feature maps. Shared weights are the weights that each neuron in a feature map has in common. A local region of the previous layer is connected to each neuron in the current layer. Local receptive fields are used to connect with local areas. After convolutional layers, pooling layers are an optional step that seeks to lower the spatial size of the feature map and to some extent control the over-fitting issue.

Let discuss about the layers in the CNN architecture:

2.1.1. Convolutional Layer

The fundamental component of a CNN is the convolutional layer. The parameters of the layer are a set of learnable filters (or kernels) that cover the entire depth of the input volume but have a narrow receptive field. Each filter is convolved across the width and height of the input volume during the forward pass. This produces a 2-dimensional activation map for each filter by computing the dot product between the filter entries and the input. As a result, the network acquires filters that turn on when it recognizes a certain kind of feature at a particular spatial location in the input.



Output [0][0] = (9*0) + (4*2) + (1*4) + (1*1) + (1*0) + (1*1) + (2*0) + (1*1)
= 0 + 8 + 1 + 4 + 1 + 0 + 1 + 0 + 1
= 16
Input image
Filter
Output array

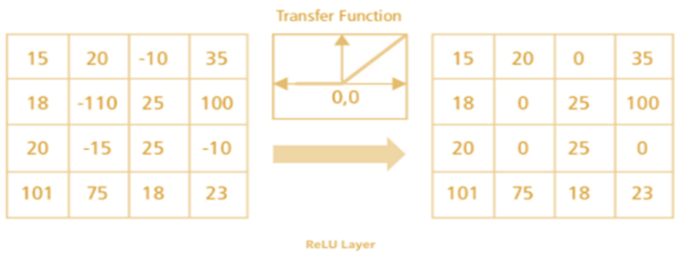
Fig 2. Convolutional Layer

2.1.2. Relu (Rectified Linear Unit) Layer

ReLU is the abbreviation of rectified linear unit , which applies the non-saturating activation function.

$$F(x) = \max(0,x)$$

By setting negative values to zero, it effectively eliminates them from an activation map. Without changing the receptive fields of the convolution layers, it causes nonlinearities to the decision function and the entire network.



15	20	-10	35
18	-110	25	100
20	-15	25	-10
101	75	18	23

15	20	0	35
18	0	25	100
20	0	25	0
101	75	18	23

Fig 3. Relu Layer

2.1.3. Max-pooling Layer

Pooling, a type of non-linear down-sampling, is another crucial idea in CNNs. The most popular non-linear function for pooling implementation is max pooling. It divides the input image into a number of rectangles and outputs the maximum for each of these sub-regions.

It makes sense that a feature's approximate placement in relation to other features is more significant than its precise location. Convolutional neural networks use pooling because of this theory. With the help of the pooling layer, overfitting can be controlled by gradually reducing the spatial size of the representation, the number of parameters used, the memory footprint, and the amount of computation required.

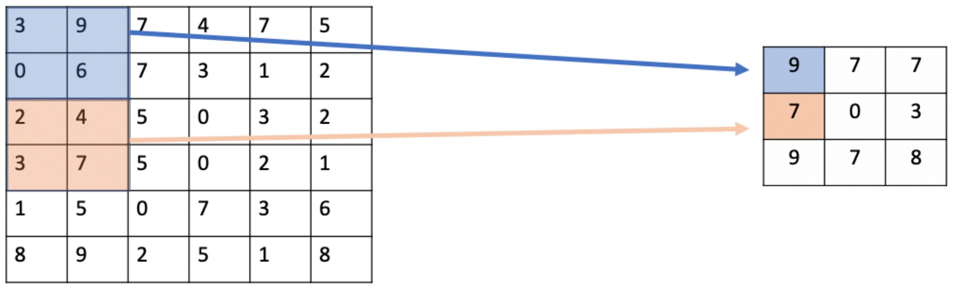


Fig 4. Max-pooling Layer

3. Numerical Results

The results in this project indicate that running a Deep learning neural network like CNN using a GPU is producing a far better speedup than running the same CNN program in a serial process.

Let us see the time taken by a serial process also considered as CPU and compare it with the time taken by a GPU to run the same CNN program.

	CPU, T(s)	GPU, T(p)
Time Taken (in sec)	82.2937 seconds	48.1508 seconds

Table 1. CPU Vs GPU

Speedup: Speedup achieved by an algorithm is defined as the ration of the Time required by the best sequential algorithm to solve a problem, T(s), to the time required by a GPU runtime to solve the same problem, T(p).

Speedup = Time taken by CPU, T(s) / Time taken by GPU, T(p)

Speedup = 82.2937 / 48.1508

Speedup = 1.709

As we can see that GPU takes also half of the time to run a CNN program compared to a CPU. That is the reason why the Speedup is almost doubled i.e., 1.709

4. Conclusion

Due to the high computational cost of training CNNs on big data sets, a flurry of open-source GPU implementation research. The purpose of this work is to help practitioners choose the best CNN implementations for various scenarios, to offer insights and recommendations to practitioners, to highlight specifics for researchers who are interested in convolution optimization on GPU, and to assist practitioners in their decision-making.

We have utilized the Convolutional Neural Network principle in this research. Additionally, we discovered how to speedup the fundamental CNN algorithm by 1.709. In order to boost performance, we also learned how to run our code on a GPU.

4. References

1. Tensor flow : www.tensorflow.org
2. Python3: www.python.org
3. Platform(Google colab): https://colab.research.google.com/drive/11xo8PF4kwzHTh5QbIfewY4N6PE_FrIHc#scrollTo=xufv5GJPZpen
4. Performance Analysis of GPU-based Convolutional Neural Networks, https://www2.seas.gwu.edu/~howie/publications/GPU-CNN-ICPP16.pdf

```
In [ ]: ## Appendix
```

```
In [ ]: import pandas as ps
        import numpy as np
        import matplotlib as plt
        import time
```

```
In [ ]: # importing TensorFlow
        import tensorflow as tf
        import keras

        # loading the fashion mnist data
        fashion_mnist = tf.keras.datasets.fashion_mnist
```

```
In [ ]: # Splitting the dataset into testing and training parts
        (train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

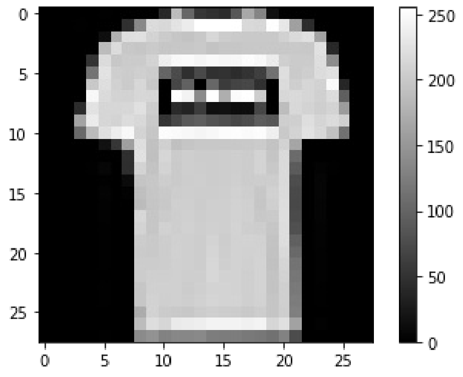
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
```

```
In [ ]: #shape of dataset
        print(train_images.shape)
        print(train_labels.shape)
```

```
(60000, 28, 28)
(60000,)
```

```
In [ ]: # importing the modules
        import matplotlib.pyplot as plt
        import numpy as np

        #ceating columns and rows
        columns = 5
        rows = 5

        # fixing the size of plot
        fig = plt.figure(figsize=(8, 8))

        # using for loop to iterate
        for i in range(1, columns * rows+1):
            data_idx = np.random.randint(len(train_images))
            img = train_images[data_idx].reshape([28, 28])
            fig.add_subplot(rows, columns, i)
            plt.imshow(img, cmap='gray')
            plt.axis('off')
        plt.tight_layout()
        plt.show()
```

In [ ]:
```
# plotting one image frrom the data
plt.figure()
plt.imshow(train_images[1], cmap='gray')

# printing the color bar
plt.colorbar()
plt.grid(False)
plt.show()
```



In [ ]:
```
# divide by 255 to range from 0 to 1
train_images = train_images / 255.0
test_images = test_images / 255.0
```

In [ ]:
```
# initializing the model
model = tf.keras.Sequential([
    # flattening the layers to have an image size of 28x28
    tf.keras.layers.Flatten(input_shape=(28, 28)),

    # Adding dense layer with 128 nodes
    tf.keras.layers.Dense(128, activation='relu'),

    # adding output  layer with 10 nodes
    tf.keras.layers.Dense(10)
])
```

In [ ]:
```
# compiling the model
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

In [ ]:
```
# training the model
start = time.time()
model.fit(train_images, train_labels, epochs=10)
end = time.time()
```

```
print("Total time taken:",end - start)
```

```
Epoch 1/10
1875/1875 [==============================] - 11s 5ms/step - loss: 0.4997 - accuracy: 0.8271
Epoch 2/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3779 - accuracy: 0.8637
Epoch 3/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3382 - accuracy: 0.8760
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3142 - accuracy: 0.8849
Epoch 5/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2949 - accuracy: 0.8920
Epoch 6/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2826 - accuracy: 0.8950
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2674 - accuracy: 0.9011
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2593 - accuracy: 0.9030
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2487 - accuracy: 0.9065
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2402 - accuracy: 0.9100
Total time taken: 42.16038656234741
```

In [ ]:
```
# finding the test accuracy
test_acc = model.evaluate(test_images,  test_labels)

# printing the accuracy
print('Test accuracy:', test_acc[1])
```

```
313/313 [==============================] - 0s 1ms/step - loss: 0.3549 - accuracy: 0.8738
Test accuracy: 0.8737999796867371
```

In [ ]:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js