Processamento Paralelo para Treinamento de Deep Neural Networks

Alisson Hayasi da Costa[1] 726494, Theodosio Banevicius[1] 619825

[1]Departamento de Computação
Universidade Federal de São Carlos (UFSCar)

1. Problema

As Redes Neurais Artificiais (RNAs) são sistemas computacionais inspirados no sistema nervoso humano cujo objetivo é simular a capacidade de aprendizado humana. São formadas por unidades de processamento, denominadas neurônios artificiais, que realizam o processamento e computação de funções matemáticas. Esses neurônios, são interligados através de conexões com pesos associados que simulam as sinapses biológicas [Haykin 2009, Gama et al. 2011].

O campo do *Deep Learning* (DL), é um sub-campo do Aprendizado de Máquina (AM) que faz uso dos modelos e técnicas tradicionais das RNAs, porém, com abordagens e estratégias mais poderosas, ampliando as capacidades de abstração e generalização das RNAs. Podemos definir o termo *Deep Learning*, como, um conjunto de técnicas e classes de algoritmos de aprendizado de máquina que, utilizam de múltiplas camadas de unidades de processamento, não lineares e organizados de forma hierárquica, resultando em um aprendizado com diversos níveis de representação e abstração [Goodfellow et al. 2016, Deng and Yu 2014].

A grande capacidade das *Deep Neural Networks* (DNNs) de lidar com imensos volumes de dados fez delas ferramentas essenciais para múltiplas áreas do conhecimento, principalmente na era *Big Data*. Contudo, não apenas desenvolver a melhor rede para uma tarefa em questão é um trabalho difícil, como também, custoso. Isso porque, para aprimorar o desempenho de uma DNN devesse encontrar uma combinação de hiperpârametros na qual o erro do modelo naquela tarefa se torne mínimo.

Uma boa combinação pode ser encontrada através de um processo conhecido como *tuning*. Isto é, vários hiperparâmetros são definidos, ajustados e seus resultados são comparados constantemente em cada treinamento a fim de encontrar as melhores ou melhor a combinação de hiperparâmetros e valores possível [Goodfellow et al. 2016, Deng and Yu 2014]. Entretanto, DNNs projetadas para dados em larga escala possuem um imenso número de neurônios, tornando cada vez mais custosa a solução do problema. Em alguns casos, o treinamento de certos modelos podem alcançar dias ou até mesmo semanas [Park 2009, Ben-Nun and Hoefler 2018]

Portanto, a fim de possibilitar o uso de estratégias de treinamento tão importantes, apresentamos nesse trabalho os benefícios da computação paralela no treinamento de redes neurais profundas.

2. Solução

A programação concorrente permite a execução de processos de maneira simultânea. Há diversas formas de se implementar a programação concorrente e paralela para otimizar o tempo de treinamento de redes neurais, as três principais formas são *Data Parallelism*, *Model Parallelism* e *Task Parallelism* [Huqqani et al. 2013, Ben-Nun and Hoefler 2018, Park 2009].

O *Data Parallelism* consiste em dividir os dados em diferentes módulos. No caso dos modelos de *Deep Learning*, estes dados podem ser o conjunto de treinamento que é dividido entre vários modelos de mesma configuração. Portanto, cada *thread* possui os mesmos modelos, porém, dados específicos. Tal estratégia geralmente é implementada para computação de matrizes ou estimativa de parâmetros em, principalmente, redes neurais convolucionais [Park 2009, Ben-Nun and Hoefler 2018].

Já o *Model Parallelism* consiste em enviar os mesmos dados para diferentes módulos, onde cada módulo possui uma parcela diferente do modelo. Portanto, cada *thread* realiza uma computação, porém, todas as *threads* compõem um único modelo. Tal estratégia geralmente é implementada para estimar o comportamento de diversos modelos em diferentes configurações simultaneamente [Huqqani et al. 2013].

Por fim, o *Task Parallelism* é semelhante ao *Model Parallelism*, contudo, ao invés de cada *thread* possuir uma parcela diferente do modelo, ela compreende um modelo totalmente diferente. Tal estratégia geralmente é implementada quando o objetivo é acelerar o treinamento de redes neurais profundas e realizar comparações de desempenho (*Deep Learning*) [Park 2009]. Portanto, nesse trabalho, utilizaremos a estratégia de *Task Parallelism*, afinal, o objetivo é otimizar o treinamento de *Deep Neural Networks*.

3. Resultados

Como estratégia adotada neste trabalho foi de *Task Parallelism*, cada modelo é treinado e testado em uma *thread* específica. Para evitar problemas durante a execução do código e o programa continuar trabalhando ao mesmo tempo que as redes são treinadas, foi um utilizado um método para garantir que o programa principal só continue em execução após o treinamento das *threads* terem sido finalizados (ou seja, o treinamento ter sido concluído).

Os experimentos foram feitos utilizando a linguagem de programação *Python 3.6.5* em conjunto com as bibliotecas de aprendizado de máquina e científica *keras* (v2.1.5), *TensorFlow* (v1.8.0), *scikit-learn* (0.19.1), *numpy* (v1.14.5) e as biblioteca padrão *threading*, *time* e *os* da linguagem.

```
1 from keras.datasets import mnist
2 from keras.utils    import to_categorical
3 import tensorflow as tf
4 import numpy, time, os, threading
```

O programa possui dois arquivos principais. O arquivo `main.py` que contém o fluxo principal do programa, carrega os dados, cria as threads e realiza não apenas o processo de treinamento, como também de teste. E, o arquivo `model.py` que cria em forma de classes os diferentes modelos, assim como, as operações correspondentes.

No arquivo `main.py`, há 4 módulos.

- `load_data()`: responsável por carregar os dados utilizados para o treinamento e teste das redes neurais. Neste caso, os dados utilizados são as imagens de dígitos manuscritos do banco de dados MNIST.
- `training_process()`: responsável por criar as threads de treinamento, executá-las e coordená-las. Neste método, é feito tanto o treinamento em múltiplas threads como em uma única thread, a fim de comparar a diferença de tempo.
- `test_process()`: responsável por criar as threads de teste, executá-las e coordená-las. Neste método, assim como no `training_process()` é feito tanto o teste em múltiplas threads como em uma única thread, a fim de comparar a diferença de tempo.
- `main()`: módulo principal do programa que realiza tanto a instanciação dos modelos quanto a chamada dos outros métodos.

3.1. Método main()

O método `main()` começa definindo uma seed padrão para reprodutibilidade nos experimentos. Em seguida, é chamado o método `load_data()` que carrega os dados do *MNIST* e retorna os exemplos (`x`) e as classes dos respectivos exemplos (`y`) para o treinamento (`x_train, y_train`) e teste (`x_test, y_test`). Após, são instanciados os três modelos implementados no arquivo `model.py`.

```
1 numpy.random.seed(1)
2 x_train, y_train, x_test, y_test = load_data()
3
4 small_MLP  = SmallMLP()
5 medium_MLP = MediumMLP()
6 big_MLP    = BigMLP()
```

Os modelos adotados são Deep Feedforward Networks com três camadas intermediárias e uma de saída. O número de neurônios em cada camada é apresentado na Tabela 1

Modelos	Neurônios nas Intermediárias	Neurônios na Saída	Total Neurônios
SmallMLP	196	10	598
MediumMLP	392	10	1186
BigMLP	784	10	2352

Tabela 1. Dimensão das *Deep Feedforward Networks* implementadas

Logo após a instanciação dos modelos, é utilizado o método de objeto `train_and_test_operations.py` que retorna as operações de treinamento e teste dos modelos.

```
small_MLP_train,  small_MLP_test  = small_MLP.train_and_test_operations()
medium_MLP_train, medium_MLP_test = medium_MLP.train_and_test_operations()
big_MLP_train,    big_MLP_test    = big_MLP.train_and_test_operations()
```

As operações são estão colocadas em listas correspondentes que, por sua vez, são enviadas como parâmetros para os métodos `training_process()` e `test_process()`, respectivamente.

```
training_process(train_operations, x_train, y_train)
test_process(test_operations, x_test, y_test)
```

3.2. Métodos training_process() e test_process()

Tanto o método `training_process()` quanto o `test_process()` possuem um comportamento comum. Ou seja, dividem os processos de treinamento e teste de cada modelo em *threads*, iniciam as *threads* e fazem a chamada do método `join()` para garantir que o programa principal só continue executando após todas as *threads* acabarem (i.e, o treinamento de todos os modelos ter sido concluído) e, assim, permitir uma comparação justa entre o custo computacional em várias *threads* e única *thread*.

```
def training_process(train_operations, x_train, y_train):
    # [...]
    train_threads = []
    for train_operation in train_operations: # Define as threads
        train_threads.append(threading.Thread(target = train_operation,
                                              args   = (x_train, y_train)))
    # Executando em várias threads
    for train_thread in train_threads:
        train_thread.start() # Inicia a execução das threads
    for train_thread in train_threads:
        train_thread.join()  # Aguarda a finalização das threads

    # Executando um única thread
    train_operations[0](x_train, y_train)
    train_operations[1](x_train, y_train)
    train_operations[2](x_train, y_train)

    # [...]
```

```
def test_process(test_operations, x_test, y_test):
    # [...]
```

```
    test_threads = []
    for test_operation in test_operations: # Define as threads
        test_threads.append(threading.Thread(target  = test_operation,
                                             args    = (x_test, y_test)))
    # Executando em várias threads
    for test_thread in test_threads:
        test_thread.start() # Inicia a execução das threads
    for test_thread in test_threads:
        test_thread.join()  # Aguarda a finalização das threads

    # Executando em única thread
    test_operations[0](x_test, y_test)
    test_operations[1](x_test, y_test)
    test_operations[2](x_test, y_test)

    # [...]
```

Executando todos os três modelos em 256 épocas de treinamento e tamanho de lote (*batch_size*) igual a 128, resultados significantes foram obtidos. Como mostra a Tabela 2, o treinamento das redes neurais em diferentes *threads* possui um tempo consideravelmente menor do que o treinamento em uma única *thread*, comprovando a ideia de que treinar múltiplas redes neurais profundas em diferentes *threads* é muito mais vantajoso do que realizar o treinamento em apenas uma única *thread*. Além disso, a mesma conclusão pode ser feita para o processo de teste.

Estratégia	Tempo de Treinamento	Tempo de Teste
Threads Separadas	78 minutos e 17.81 segundos	1 minuto e 38.41 segundos
Thread Única	94 minutos e 32.29 segundos3	1 minuto e 47.02 segundos

Tabela 2. Tempo de execução dos modelos em diferentes estratégias

Referências

[Ben-Nun and Hoefler 2018] Ben-Nun, T. and Hoefler, T. (2018). Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *CoRR*, abs/1802.09941.

[Deng and Yu 2014] Deng, L. and Yu, D. (2014). *Deep Learning: Methods and Applications*, volume 7.

[Gama et al. 2011] Gama, J., Faceli, K., Lorena, A., and De Carvalho, A. (2011). *Inteligência artificial: uma abordagem de aprendizado de máquina*. Grupo Gen - LTC.

[Goodfellow et al. 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[Haykin 2009] Haykin, S. S. (2009). *Neural networks and learning machines*. Pearson Education, Upper Saddle River, NJ, third edition.

[Huqqani et al. 2013] Huqqani, A. A., Schikuta, E., Ye, S., and Chen, P. (2013). Multicore and gpu parallelization of neural networks for face recognition. *Procedia Computer Science*, 18:349 – 358. 2013 International Conference on Computational Science.

[Park 2009] Park, S. J. (2009). An analysis of gpu parallel computing. pages 365–369.