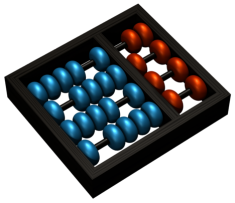
Deep-Learning Rede Neural Convolucional Paralelizada em PyCUDA

M0644 - Programação Paralela
Prof.: Dr. Guido Araújo

Paulo Ricardo Finardi, ra144089
Lucas Oliveira David, ra188972



UNICAMP

Campinas, 1 de novembro de 2017

1 Introdução

Em aprendizado de máquina, redes neurais possuem sem dúvida um lugar de destaque. Popularizadas na década de 80, as *Multi-layered Perceptron* (MLP ou redes *fully connected*) foram extensivamente aplicadas nos mais diversos problemas, devido aos seus modelos altamente flexíveis e precisos. Entretanto, as exigências de espaço e processamento (muitas vezes extremas) fizeram com que a utilização destas fosse impraticável para problemas maiores, como o reconhecimento de padrões em imagens de alta resolução. Para isso, um grande número de variações foram desenvolvidas ao longo dos anos. Em especial, vale lembrar das Redes Neurais Convolucionais (em inglês *Convolutional Neural Network* (CNN)), onde ao menos uma das camadas consiste na aplicação da operação de convolução do sinal de entrada com um ou mais filtros (ou *kernels*) [1] que foram iterativamente aprendidos. Tal propriedade faz com que CNNs sejam modelos ideais no processamento de imagens ou vídeos, sendo amplamente utilizadas nestas tarefas. A Figura 1 exemplifica a arquitetura de uma rede com duas camadas convulacionais e duas *fully connected* a fim de lidar com o problema da classificação de dígitos.

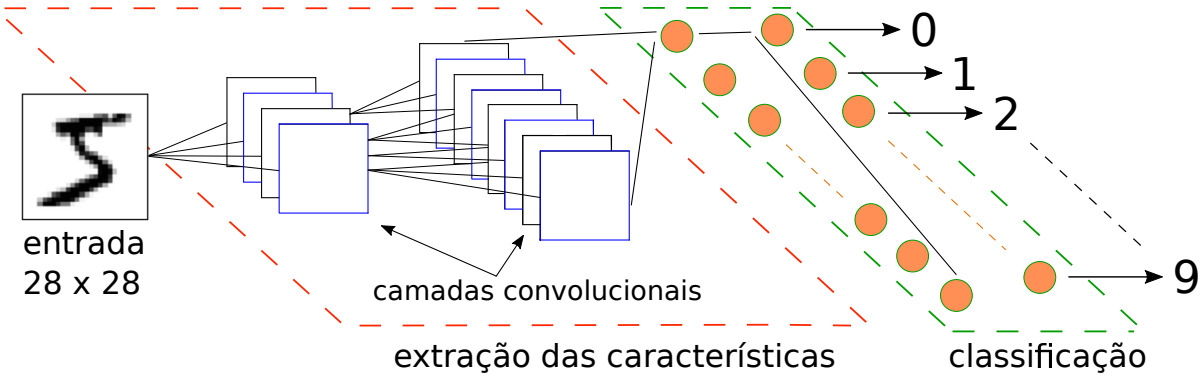


Figura 1: Arquitetura de uma rede neural convolucional.

No que diz respeito à performance, redes neurais apresentam um grande gargalo de processamento em suas fases de treino devido ao grande número de operações matriciais ali executadas. Atualmente, é comum que indivíduos treinem redes complexas sobre grandes conjuntos de dados, requerindo dias para convergência. Há portanto um grande interesse em otimizar o processo de treinamento das redes.

1.1 Motivação

Uma CNN se aproveita de três ideias importantes que podem ajudar a melhorar um sistema de aprendizagem de máquina [2]: interações esparsas, partilha de parâmetros e representações equivariantes. Além disso, a convolução permite tratar problemas com entradas de tamanho variável. Uma rede neural convencional utiliza a multiplicação de matrizes por uma matriz de parâmetros entre cada unidade de entrada e saída. Assim, cada unidade de saída interage com cada unidade de entrada. CNNs possuem interacções esparsas, isto é conseguido pelo uso de núcleos menores do que a entrada. Portanto, ao processar uma imagem a imagem de entrada pode ter milhares de pixels, mas podemos detectar pequenas e siginificativas características com núcleos que que ocupam apenas dezenas ou centenas de pixels.

Partilha de parâmetros é o uso do mesmo pa-

râmetro para mais de uma função em um modelo. Numa rede neural tradicional, cada elemento da matriz de peso é usado apenas uma vez. Como sinônimo de partilha de parâmetros, pode-se dizer que uma rede "amarrou pesos", porque o valor do peso aplicado a uma entrada está ligada ao valor de um peso aplicado em outros lugares.

Em matemática, uma aplicação invariante é uma função entre dois conjuntos que comuta com a ação de um grupo. Especificamente, seja G um grupo e sejam X e Y os $G-$conjuntos associados. Uma função $f : X \mapsto Y$ é dita equivariante se $f(g \cdot x) = g \cdot f(x)$.

2 Perfilação

Realizamos a análise dinâmica do programa utilizando a ferramenta cProfile, padrão para programas gerados com a linguagem Python. Em seguida, o utilitário KCacheGrind foi empregado na visualização do relatório gerado pelo *profiler*. Pela Figura 3, constata-se que a chamada `fit` é de fato o gargalo do processamento, ocupando cerca de 98% do tempo total de execução.

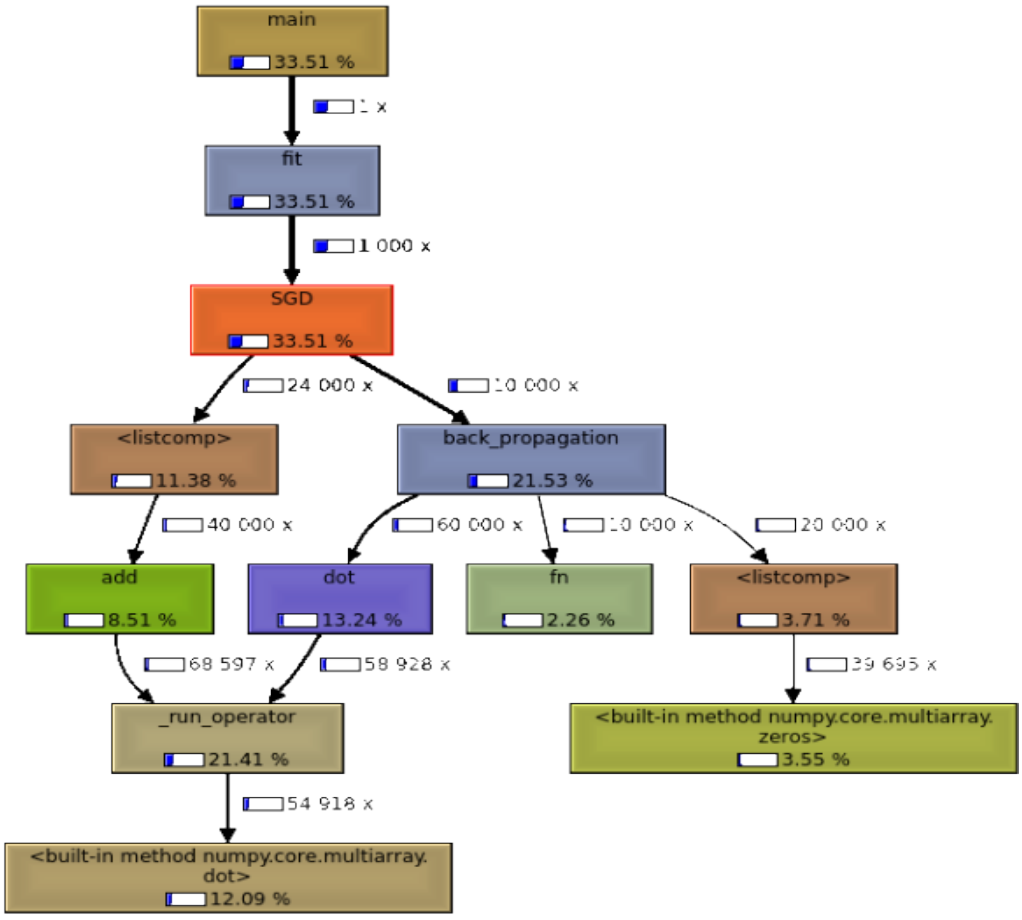
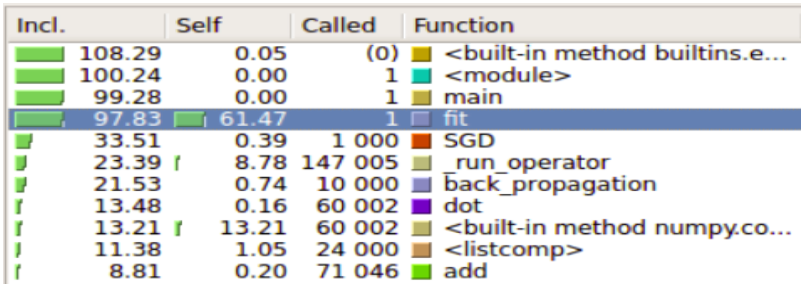


Figura 2: Árvore de perfilação como exibida pelo KCachegrind



Incl.	Self	Called	Function
108.29	0.05	(0)	<built-in method builtins.e...
100.24	0.00	1	<module>
99.28	0.00	1	main
97.83	61.47	1	fit
33.51	0.39	1 000	SGD
23.39	8.78	147 005	_run_operator
21.53	0.74	10 000	back_propagation
13.48	0.16	60 002	dot
13.21	13.21	60 002	<built-in method numpy.co...
11.38	1.05	24 000	<listcomp>
8.81	0.20	71 046	add

Figura 3: Relatório de perfilação exibido através do KCachegrind.

A Figura 2 apresenta o grafo de execução do programa, onde a largura de um determinado arco indica qual dos caminhos tomou mais tempo de execução. Com base em ambas as figuras, decidiu-se pelo enfoque na paralelização das operações da função `fit`.

3 Estratégia de paralelização

Embora os processos de treinamento e execução de redes neurais sejam quase por completo compostos por operações matemáticas elementares, a pluralidade de arquiteturas destas dificulta a criação e utilização de um modelo de paralelização definido, como ocorreu em vários dos projetos ao longo da disciplina. Além disso, havia do grupo o interesse em abordar, mesmo que de maneira simples, conceitos interessantes como abstração de complexidade e computação heterogênea. Com isso em mente, o paralelismo foi implementado de maneira implicita:

- Foi criado o módulo `operators`, contendo um mapa de operações matemáticas básicas, como a adição, multiplicação de matrizes e convolução, além de contratos para estas operações que meramente apontam para o mapa e executam as operações ali definidas. Finalmente, o módulo também conta com um procedimento `set_mode`, capaz de alterar o mapa atual para um dos três possíveis: `_sequential`, `_vectorized` e `_cuda`.
- Três sub-módulos de mesmo nome que os modos em `operators` foram implementados, onde cada um possui definições distintas para as operações mencionadas acima. Enquanto o `sequential` contém operações definidas de forma trivial e o `vectorized` utiliza-se de chamadas da biblioteca NumPY para operações vetoriais, o módulo `cuda` utiliza a biblioteca PyCUDA [3] para compilar kernels implementados em C++ e carregá-los para a utilização do usuário.

Usuários podem então definir um modo de operação desejado utilizando escopos definidos pela classe `Device` e simplismente executar as operações definidas em `operators`, automaticamente paralelizando (ou serializando) o código executado. No trexo de código abaixo, o usuário define operações de convolução e adição que serão executadas na GPU, além de *dot* e adição que serão executadas vetorialmente na CPU:

```
with Device('gpu'):
    W = op.add(op.conv(X, Kernels))

with Device('vectorized'):
    y = op.add(op.dot(W, X), b)
```

A Figura 4 esquematiza a estratégia de paralelização e a Figura 5 mostra a taxa de acerto e a relação de *speed-up* de uma rede *fully connected*.

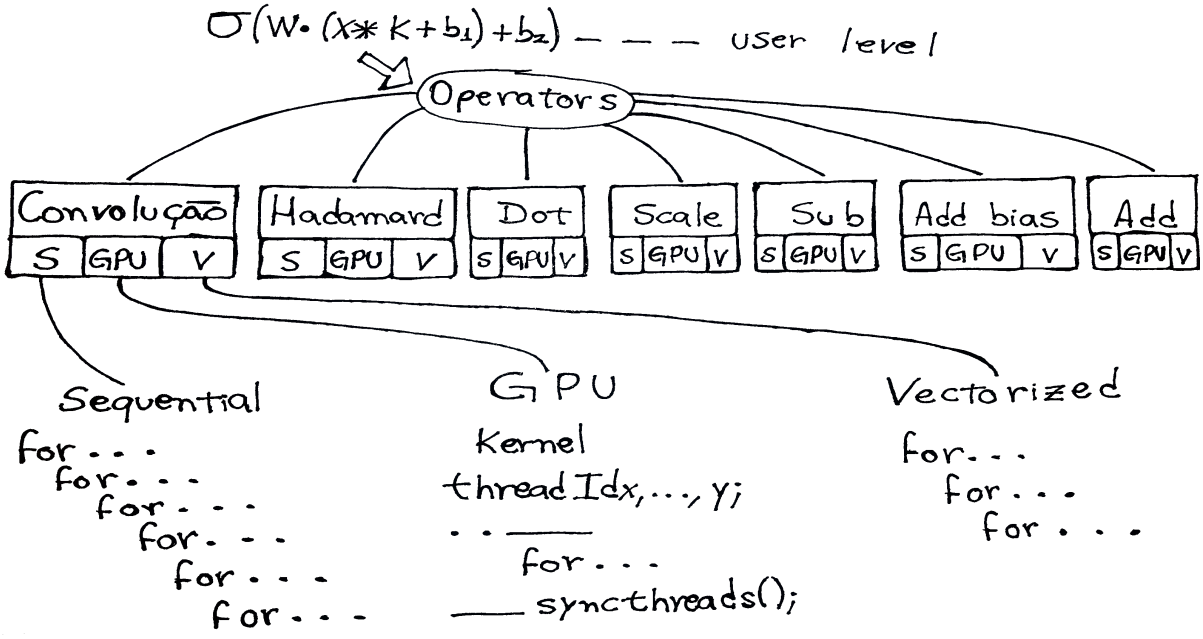
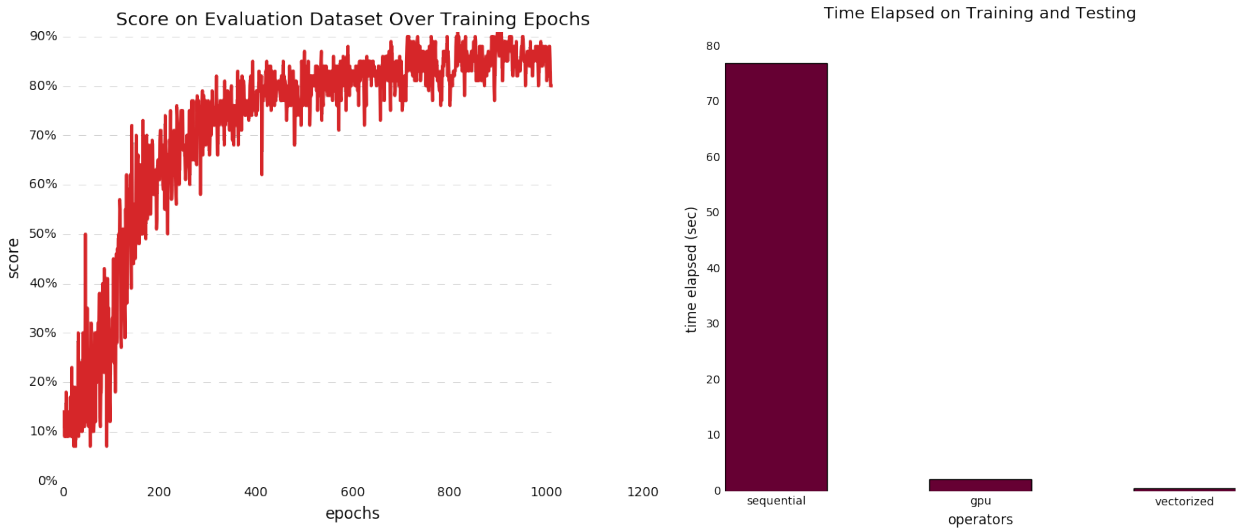


Figura 4: Estratégia de paralelização



(a) (b)

Figura 5: (a) gráfico da taxa de acerto de uma rede *fully connected*. (b) *speed-up*: *sequential*/GPU = 25.6, *sequential*/*vectorized* = 77 e *vectorized*/GPU = 0.33.

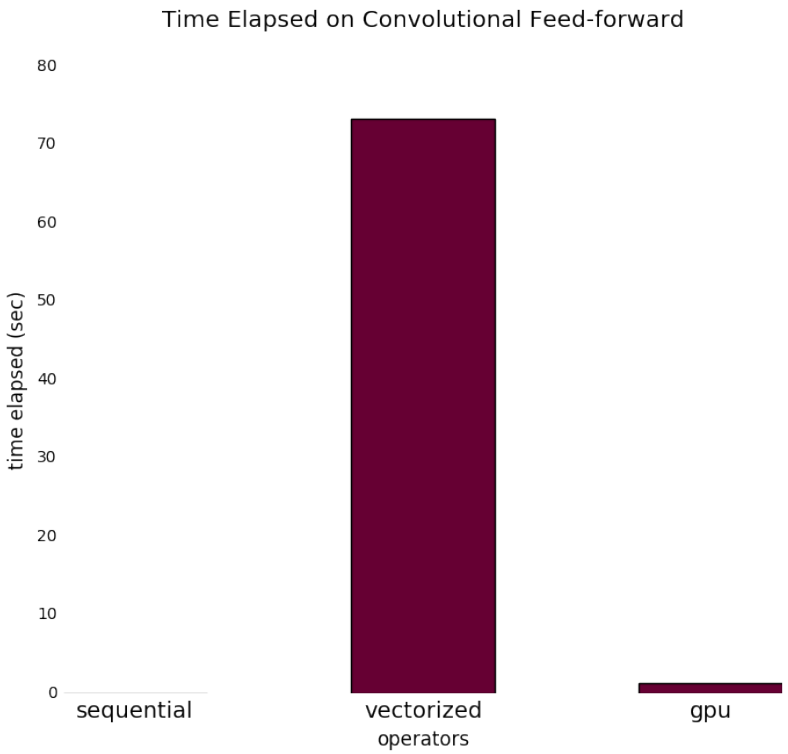


Figura 6: *speed-up vectorized*/GPU = 62.5. O tempo *sequential* não foi considerado devido a grande demora de execução.

4 Dificuldades encontradas

Os desafios deste trabalho foram muitos. Primeiramente, não possuíamos as implementações em código *sequential* e *vectorized*. Embora grande parte do módulo `vectorized` tenha sido reaproveitado da biblioteca NumPY, ambos `sequential` e `cuda` foram completamente implementados do zero.

Com relação a paralelização em CUDA, podemos citar o grande número de operadores necessários como nossa maior dificuldade. Foram nove operadores implementados, dentre os quais sete foram utilizados durante as computações relacionadas ao treinamento de redes.

Os processos de *feed-forward* em ambas redes *fully-connected* e convolucionais, bem como o algoritmo *Stocastic Gradient Descent* das *fully-connected* foram facilmente implementados. Entretanto, o grupo encontrou uma grande dificuldade no desenvolvimento do SGD aplicado a fim de treinar as redes convulacionais. Este processo está descrito pela literatura de muitas maneiras distintas, muitas vezes contradizendo uns aos outros em operações ou ordem de aplicação. Uma análise mais extensiva precisa ser empregada, a fim de se identificar qual operação resultaria, de fato, na convergência de nossos modelos.

Finalmente, encontramos uma grande dificuldade em vencer as implementações vetoriais provenientes do NumPY com os operadores `cuda`. Acreditamos que o tempo de *offload* de dados para a GPU reduz severamente a performance dos operadores `cuda`, dando a vantagem de localidade ao `vectorized`. Ademais, de acordo com [4], citamos os seguintes argumentos: o NumPY faz uso de BLAS (do inglês *Basic Linear Algebra Subroutines*) e arquiteturas *multicore*. O BLAS é uma biblioteca cuidadosamente ajustada para executar processos otimizados no hardware. Com o uso do BLAS e arquiteturas multicore o NumPY pode executar pro-

cessos em paralelo (se esta operação for mais rápida) sem termos que alterar qualquer parte do código. O tempo apresentado na Figura 5 foi executado com uma máquina que possui BLAS e arquiterura *multicore*. Dado esses pontos, os participantes deste grupo encontram mais conforto na perda de performance da implementação `cuda` em relação à `vectorized`.

Referências

[1] Neural networks and deep learning. endereço: `http://neuralnetworksanddeeplearning.com/index.html`. Acessado em 28/Jun/2016.

[2] Deep learning. endereço: `http://www.deeplearningbook.org`. Acessado em 08/Jun/2016.

[3] Gpu accelerated computing with python. endereço: `https://developer.nvidia.com/how-to-cuda-python`. Acessado em 02/Jun/2016.

[4] Parallel programming with numpy and scipy. endereço: `http://scipy.github.io/old-wiki/pages/ParallelProgramming`. Acessado em 29/Jun/2016.