Aceleradores para Aprendizado Profundo *Deep Learning*

André Vinícius Lopes, Victor Mayrink

Monografia apresentada
ao
curso MAC5742/0219 Introdução à Programação Concorrente,
Paralela e Distribuída

Professor: Prof. Dr. Alfredo Goldman
Monitor: Pedro Henrique Rocha Bruel

São Paulo, 2017

Aceleradores para Aprendizado Profundo *Deep Learning*

André Vinícius Lopes - 9527845
Victor Mayrink - 10412183

Sumário

Capítulo 1

Introdução

Extrair informações de imagens não é uma tarefa trivial, na área de análise de imagens médicas, por exemplo, é interessante segmentar áreas de interesse, verificar agrupamento de o objetos, analisar formas, tais como, retas, curvas e bordas. Em geral, é necessário um especialista para fazer a análise e este terá que usar seus conhecimentos de sua área de estudo para fazer as operações necessárias na imagem, para então, extrair informações para seu estudo.

Em 1943, não havia um algoritmo para o treinamento deste modelo e os valores da matriz de pesos das redes neurais tinham de ser escolhidos e calculados manualmente. Devido a este motivo, as redes neurais caíram em desuso. Somente em 1958, Rosenblatt, F. F. apresentou um algoritmo de treinamento, chamado de *backpropagation*, o qual automatizou este processo, no entanto, na época não havia processadores e placas aceleradoras GPGPU'S com poder de processamento suficiente para executar este algoritmo em um tempo razoável. Ao aumentar o tamanho da rede neural, o tempo de treinamento para se ter um bom resultado era tão grande que usar este modelo se tornava inviável. Portanto, novamente, caiu em desuso.

Em 2006, Hinton, G. apresentou uma nova maneira de realizar o treinamento de redes neurais usando GPGPU's. A aceleração no tempo de execução fez com que este modelo ficasse em evidencia na comunidade cientifica. A NVIDIA, com seu pacote CUDA Nvidia [2010] para programação paralela em GPU's , permitiu que muitos cientistas criassem pacotes para modelar redes neurais e treiná-las de forma paralela.

Importante notar que, nas literaturas sobre aprendizado profundo (*deep learning*), em geral, se refere a redes neurais artificiais somente pelo termo *redes neurais*.

Portanto, a pesquisa em aceleradores para *deep learning* é importante para aumentar a eficiência desta área de pesquisa.

Capítulo 2

Conceitos de Aprendizado de Máquina

Tom Mitchell definiu aprendizagem de máquina da seguinte maneira : "É dito que um programa de computador aprende de uma experiencia E em relação a uma tarefa T e com medida de desempenho P, se seu desempenho em T, medido por P, melhora com a experiencia E",(Tom Mitchell, 1997, p. 2, minha tradução) Mitchell [1997]

Entre os exemplos de problemas que são tratados com aprendizado de máquina, podemos citar o problema de classificar espécies de peixes em imagens marítimas.

Dado um problema de classificar espécies de peixes em imagens marítimas, pode-se definir os componentes do aprendizado. X são os dados de entrada formado pelas imagens marítimas, Y é a classificação do peixe correspondente a X, ou seja, o conjunto de classes em que X pode ser classificado. $f : X \to Y$ é a função-alvo e D, o conjunto formado pelos dados de entrada e as saídas correspondentes, ou seja, $D = \{x_{1..n}, y_{1..n}\}$, sendo n, o número de exemplos.

Define-se aprendizado supervisionado, dado um conjunto de dados D formado por X e Y, ou seja, os exemplos e os rótulos respectivos.

Além disso, temos o algoritmo de aprendizado que utiliza D para escolher uma função g que aproxima f. Este algoritmo, escolhe g a partir de um conjunto de hipóteses H, formado por um conjunto de candidatos que aproximam f. O algoritmo de aprendizado tem como objetivo, escolher g, de tal maneira que essa seja a melhor hipótese de H, a qual se aproxima mais de f, ou seja, com erro mínimo para um medidor de erro P em D.

Possibilidade de aprendizado

Sendo D um conjunto de dados de treinamento, usado para estimar g de um conjunto de hipóteses H, podemos afirmar que g foi um bom resultado do algoritmo de aprendizado se D fez g inferir informações de dados que não estão presentes em D.

Apesar de haver os valores de g para D, desconhece-se g para exemplos que não estão presentes em D.

Para verificar qual desempenho o modelo g tem, há de se verificar o quanto g erra e acerta para um conjunto de dados que não foi usado no treinamento. Para tanto, é necessário ter um outro conjunto de dados que não influenciaram na busca por g em H. Em geral, este conjunto usado para verificar o desempenho de g, é chamado de conjunto de teste.

Portanto, define-se que E_{in} é a quantidade de erro em g do conjunto de dados usado para treinamento, ou seja, a aprendizagem de g, e E_{out} é a quantidade de erro de g em dados que estão no conjunto de teste e não estão no conjunto de treinamento.

Seja N a quantidade de exemplos de um conjunto de dados e $E[\bullet]$ a esperança de $\bullet$, define-se matematicamente :

$$E_{in}(h) = \frac{1}{n}\sum_{n=1}^{n}[g(X) \neq f(X)]$$

$$E_{out}(h) = E[g(X) \neq f(X)]$$

2.1 Funções de perda

E_{in} e E_{out} são calculados como uma função de perda, a qual é utilizada para medir a eficiência do classificador ou de um modelo de regressão. Porém, se esta função não é diferenciável, muitos algoritmos que necessitam desta propriedade para poder otimizar esta função, não poderão ser utilizados. Entre estas funções que necessitam dessa propriedade, a mais utilizada para os modelos de redes neurais artificiais é chamada de gradiente descendente.

Há diversas funções de perda que podem ser utilizadas, algumas são apenas para classificação e outras para problemas de regressão. Como exemplo de funções de perda, pode ser citado a entropia cruzada categórica e entropia cruzada binária, as quais servem para problemas de classificação

2.2 Modelos Lineares

Um modelo de aprendizagem linear pode ser utilizado para problemas de classificação e de regressão.Um classificador linear gera combinações lineares dos atributos de $x \in D$ para separar a amostra em duas classes.Sendo h a hipótese escolhida pelo modelo, w o vetor coluna formado por X , e $X = 1xR^{+1}$ pois, $x_0 = 1$ devido a adição da variável 'bias', também chamada de w_0. A adição da dimensão 'bias' permite aumentar o espaço de entrada. Sem este termo, seria apenas uma multiplicação de um vetor de entrada com uma matriz.

$$h(x) = \pm(w^T x)$$

2.2.1 Perceptron

O perceptron é um modelo matemático de um neurônio e é muito eficiente para resolver problemas que são linearmente separáveis.A entrada é a quantidade de variáveis presentes em x_n, sendo $x_n \in X$, e sua saída são dois valores distintos. Seja L, um limiar e w o conjunto de pesos dado a cada coordenada de x, o perceptron retorna um valor N se $\sum_{i=1}^{d}(w_i x_i) > L$, ou retorna um valor M se $\sum_{i=1}^{d}(w_i x_i) < L$. Em geral, N é o valor positivo 1, e M é -1.

Redes Neurais Artificiais

O modelo de redes neurais artificiais foi apresentado em 1943 por McCullock, W. e Pitts, McCulloch e Pitts [1943]. Este modelo é inspirado no cérebro humano, mais precisamente no seu nível mais básico, o neurônio. È uma generalização do perceptron, e muito utilizado em problemas de regressão e visão computacional, e inclusive, há aplicações, para outros casos, como por exemplo, para conjunto de dados formados por sinais de áudio. Uma rede neural artificial aprende padrões e dado um conjunto de entrada, teremos um conjunto de saída. Este modelo é formado neurônios, bias e funções de ativação.

Um neurônio multiplica sua entrada por um peso w e soma as multiplicações e o valor resultante é passado para uma função de ativação ϕ. Cada w, se referente a conexão entre 2 neurônios.

$$h(x_i, w_i) = \phi(\sum_{i}^{d}(w_i x_i))$$

Este modelo, ainda pode ser dividido em camadas : de entrada, escondidas e de saída.
Uma camada é formada por 1 ou mais neurônios. A camada de entrada é responsável por aceitar os dados que vem de fora da rede neural. Cada neurônio nesta camada aceita um único dado por vez. Dependendo da escolha de função de ativação na rede neural, os dados deverão estar normalizados antes de serem passados para a camada de entrada.
A camada de saída devolve um conjunto de valores que foi estimado pela rede neural. Dependendo da função de ativação escolhida, um tipo de dado será retornado.
As camadas escondidas facilitam que a rede neural aprenda representações mais complexas dos

dados. Estas camadas apenas recebem dados de outros neurônios e enviam dados para outros neurônios.

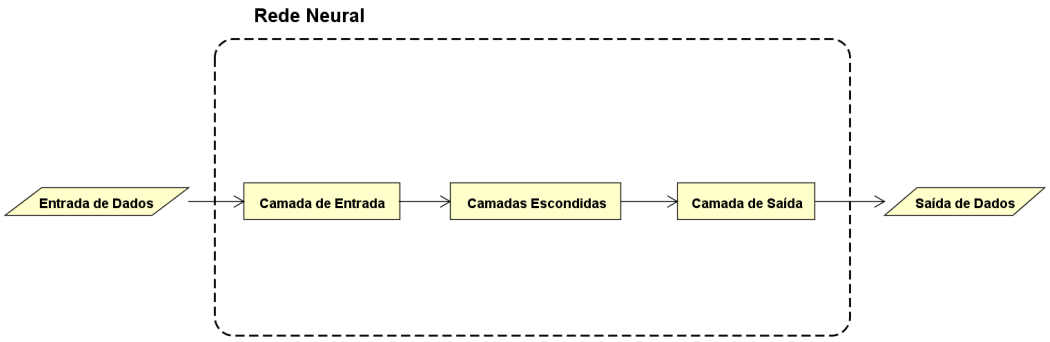
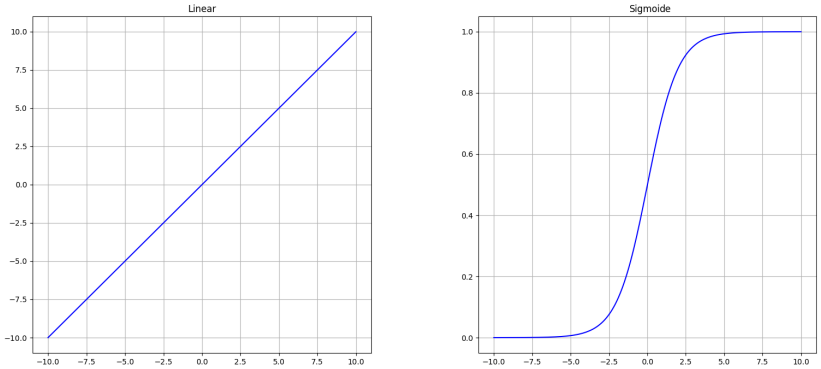


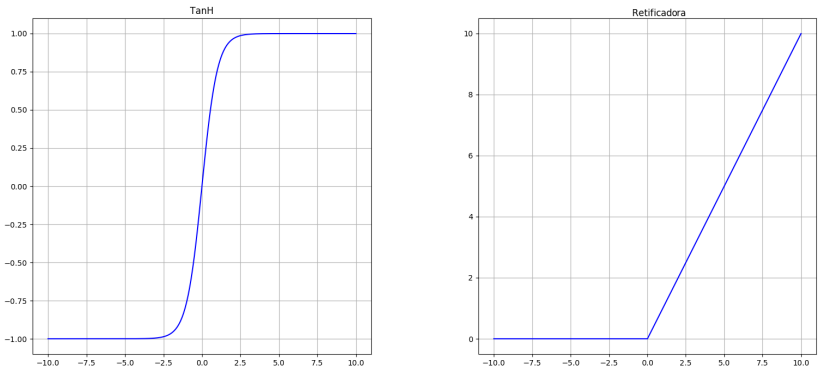
Figura 2.1: *Estrutura simplificada de uma rede neural*

Funções de Ativação

Escolher funções de ativação apropriadas é importante pois define em que formato os dados de entrada deverão estar. Ao usar funções do tipo sigmoide, por exemplo, um conjunto de dados D não deve ter valores negativos. Por outro lado, caso isto seja necessário, pode-se usar a função tangente hiperbólica. Além disso, há de se verificar qual função estará na camada de saída, pois caso seja um problema de classificação binária, por exemplo, pode-se usar a função sigmoide ou softmax.



(a) *Função Linear e Sigmoide*



(b) *Função tangente hiperbólica e retificadora linear*

Perceptron de Múltiplas Camadas

O perceptron de múltiplas camadas (PMC), é a implementação do perceptron com mais de 1 camada de neurônios. Nos casos de classificação em que os dados não são linearmente separáveis, pode-se usar este modelo pois este , ao ter mais de uma camada, permite que mais de uma reta separe os elementos. O teorema da aproximação universalCsáji [2001] diz que um PMC com uma única camada escondida com uma quantidade suficiente de neurônios, pode aproximar qualquer função continua. Este teorema, provado por George Cybenko em 1989 , foi feito usando a função de ativação sigmoide e o teorema em si, fazia suposições sobre as funções de ativações. Porem, em 1991, Kurt Hornik Hornik [1991], mostrou que este teorema é valido devido a própria estrutura de múltiplas camadas e não devido a alguma escolha de função de ativação.

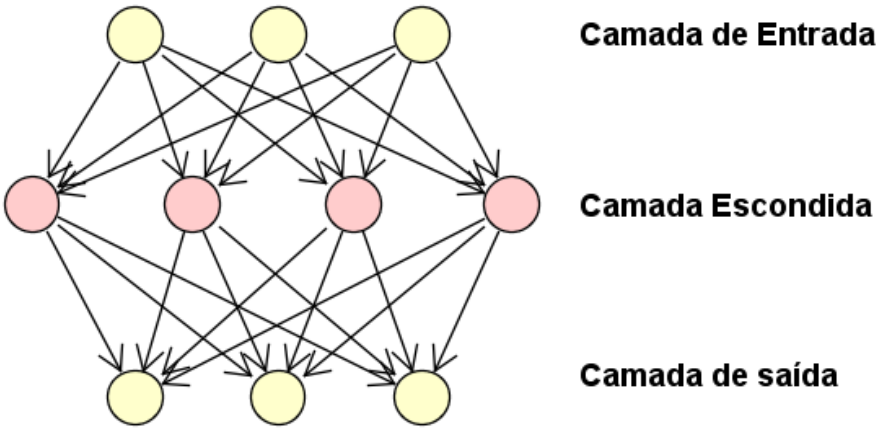


Figura 2.3: *Representação simplificada de um perceptron de múltiplas camadas*

Cada camada escondida do perceptron de múltiplas camadas é também um tipo chamado de camada densa. Uma camada densa conecta todos os neurônios da camada anterior com os da camada atual.

Treinamento

O treinamento de redes neurais envolve em se ter valores para a matriz de pesos que minimizam a função de custo. Sendo os valores de pesos números pertencentes ao conjunto dos números reais, mesmo para um modelo com poucos neurônios, há infinitas permutações possíveis. Portanto, é necessário um algoritmo que calcule valores para os pesos, de forma que, durante o treinamento, teremos cada vez mais, um menor erro de saída. Em geral, um modelo não dará como saída o valor correto de um conjunto de dados D antes do treinamento. Portanto, os pesos são ajustados para aproximar o modelo para as saídas corretas.
Os algoritmos mais eficientes para ajustar os pesos de um modelo de rede neural, utilizam o gradiente calculado pela derivada da função de erro, a saída esperada e a saída atual do modelo.

Gradiente Descendente

A derivada de uma função de custo permite calcular o gradiente para cada peso w, isto permite ajusta-los para um valor que minimiza o erro. O gradiente indica em como ajustar o valor atual de w para minimizar este erro. Se não há erro, o gradiente é 0. Se o sinal do gradiente é negativo, o valor de w deve ser aumentado e se o sinal é positivo, deve ser diminuído.
O algoritmo gradiente descendente permite ajustar todos os pesos w utilizando o gradiente calculado para cada um, utilizando os exemplos do conjunto de dados de treinamento D.

Algoritmos de retro-propagação *backpropagation*

O algoritmo de retro-propagação é utilizado com o calculo de gradiente de forma que a entrada , ou seja, o exemplo de treinamento de D é propagado por todas as camadas da rede neural, e a saída do modelo é utilizada para calcular o erro pela função de custo. Após isso, os valores dos erros são propagados de volta, começando pela camada de saída até a camada de entrada, com isto, os gradientes são calculados e os pesos ajustados. O objetivo é diminuir os gradientes para valores menores, desta forma, diminuindo o erro e aproximando os valores que saem do modelo, para os valores esperados de D.
Este algoritmo pode ser implementado em dois modos : online e em lote *batch*. No modo online, os pesos são modificados para cada exemplo de treinamento visto pela rede neural. Ou seja, para cada exemplo, é aplicado o algoritmo de retro-propagação e os pesos são ajustados.
No modo lote *batch*, dado um conjunto de treinamento D com n exemplos, temos um tamanho de lote b, tal que $b <= n$. Os pesos são modificados para cada conjunto de exemplos de tamanho b : soma-se os gradientes de cada exemplo deste conjunto e só então se ajusta os pesos w.

Sendo ϵ a taxa de aprendizagem, w cada peso da matriz de pesos da rede neural, α o momento *momentum*,considere os pesos e os gradientes calculados como um vetor unidimensional, a atualização dos pesos é feita conforme a equação abaixo :

$$\Delta_{w(t)} = -\epsilon \frac{\partial E}{\partial_{w(t)}} + \alpha \Delta_{w_{t-1}}$$

A taxa de treinamento aumenta ou diminui em escala , o gradiente. Uma taxa muito alta não permite que o modelo converga pra bons valores de pesos, e uma taxa baixa fara com que o modelo demore para adquirir bons valores de pesos.

O momento *momentum* determina a porcentagem da iteração anterior que deve ser aplicada na iteração. È um modo de permitir que o algoritmo de retro-propagação, escape do minimo local, pois este, tem como desvantagem, a tendencia de se manter em mínimos locais.

Esta equação calcula Δ, ou seja, a mudança para cada peso, como o produto do gradiente e a taxa de treinamento e adicionando o produto da mudança de peso da iteração anterior com o momento *momentum*.

Importante notar que a taxa de treinamento e o momento são hiper-parâmetros, e em geral, são escolhidos de modo empírico, o que é considerado uma desvantagem deste tipo de algoritmo de treinamento.

Gradiente Descendente estocástico

O algoritmo de gradiente descendente estocástico é um tipo de procedimento de retro-propagação, o qual também pode ser utilizado no modo online ou em lote, porém com algumas modificações. No modo online, escolhe-se os exemplos aleatoriamente para serem usados na retro-propagação. No modo de lote, o sub-conjunto, também chamado de mini-lote *mini-batch* é construído escolhendo-se elementos aleatórios do conjunto de treinamento, dado o tamanho deste conjunto. A diferença entre o gradiente descendente e o gradiente descendente estocástico é a escolha aleatória de exemplos. O modo lote não é implementado no algoritmo de gradiente descendente, apenas no gradiente descendente estocástico.
A principal vantagem é que no modo lote, ele é muito mais rápido do que o gradiente descendente. Apesar de , em geral, o GD apresentar um menor erro, pode-se aproximar deste, aumentando o número de iterações do algoritmo, e ainda assim, tende a finalizar mais rapidamente.

Capítulo 3

Aprendizado profundo *Deep learning*

Definimos aprendizado profundo, *Deep Learning*, como uma maneira de treinar qualquer rede neural profunda. Em geral, uma rede neural para ser considerada profunda tem de ter mais que duas camadas.Em 1943, Pitts McCulloch e Pitts [1943], introduziu modelo perceptron de múltiplas camadas *MLP*, desde então, podia-se criar redes neurais profundas.
Deep Learning é composto de técnicas inovativas , tais como :

- Uso de Unidades retificadoras lineares $ReLU$
- Uso de regularização com *Drop out*
- Convoluções em redes neurais, as quais são chamadas de Redes Neurais Convolucionais Lecun e Bengio [1995]

Dropout

Para diminuir o *overfitting*, os modelos de *Deep Learning* tendem a utilizar *Drop out*. Esta técnica consiste em que, alguns neurônios e suas conexões nas camadas escondidas são ignorados e não tem seus pesos alterados ou usados,periodicamente, em algumas iterações no processo de treinamento. Foi apresentada em 2012 por Hinton, Srivastava, Krizhevsky, Sutskever e Salakhutdinov Hinton *et al.* [2012], e tem como principal característica não permitir a co-adaptação de neurônios. Dado 2 neurônios, quando um deles é removido temporariamente, a rede neural terá que ajustar o outro neurônio para compensar esta remoção.
Dado isto, ajuda a mitigar a situação do modelo memorizar os dados de treinamento.
È comum que esta técnica seja implementada em forma de camada escondida. Dada uma camada escondida, pode-se ter uma camada de drop out em que cada neurônio se comporta como uma camada densa, no entanto, esta terá um hiper-parâmetro p, que é a probabilidade de um neurônio desta camada ser escolhido para ser removido temporariamente, junto com suas conexões.

Redes Neurais Convolucionais e Camadas Convolucionais

A rede neural convolucional foi desenvolvida tendo o olho biológico como inspiração e devido a seus resultados, é um modelo variante do perceptron de múltiplas camadas, muito importante na área de computação visual, pois este modelo tem um ótimo desempenho para reconhecimento e classificações de imagens. Foi apresentada pela primeira vez por Fukushima em 1980 Fukushima [1980].
Em 1998, este modelo de rede neural foi melhorado significantemente por LeCun, Bottou, Bengio e Haffner LeCun *et al.* [2001].
As redes neurais convolucionais utilizam camadas convolucionais em sua estrutura. Estas camadas tem como objetivo extrair características *features* do conjunto de dados de treinamento. O termo convolucional vem do operador de convolução, ...; Esta camada é composta de diversos hiper-parâmetros, sendo estes :

- Numero de filtros
- Dimensão dos filtros
- *Stride*
- Dimensão de *Padding*

As camadas convolucionais apresentam filtros que tem a forma de polígonos de 4 lados com ângulos retos, sendo mais comum, o formato de um quadrado. Cada filtro, também chamado de *kernel* pode ter seu tamanho configurado por um parâmetro de largura e altura. Sendo a imagem de entrada, uma matriz formada por pixeis, então cada filtro é uma matriz de tamanho menor que a imagem de entrada que transita por cada linha da imagem e calcula o produto escalar, e a matriz formada por este processo é chamada de mapa de características ou mapa de ativação. Além disso, cada filtro deve poder começar pelo lado esquerdo superior.

Sendo, I_w a largura da imagem, f a largura do filtro, p o *padding*, e s o *stride*, podemos calcular a quantidade de passos que o filtro fara na imagem de entrada da seguinte forma :

$$passos = \frac{I_w - f + 2p}{s + 1}$$

Definimos um mapa de ativação como o resultado de aplicar o filtro em todas as sub-regiões da matriz de entrada , adicionando o termo bias e aplicando uma função de ativação.
Há pesos entre a própria camada convolucional e a camada anterior, cada pixel na camada é um peso. Sendo Q_{pesos} a quantidade de pesos entre estas camadas, $Q_{filtros}$ a quantidade de filtros, f a largura do filtro e h a altura do filtro, calculamos da seguinte maneira :

$$Q_{pesos} = Q_{filtros} * f * h$$

Definimos passo (*stride*), como a quantidade de pixels que o filtro pulará a cada movimentação. Para um valor de *stride* igual a 1, o filtro ira se mover um pixel por vez para a direita e ao chegar na extremidade, um pixel para baixo. O *Stride* não pode ter valor igual a 0, senão o filtro não ira transitar.
Preenchimento (*Padding*) é uma tecnica de se adicionar valores 0's nas bordas da matriz de entrada da camada convolucional. Isto permite que o filtro não perca informação das bordas, pois a cada vez que a matriz passa por uma camada convolucional, sua dimensão diminui. *Padding* permite mitigar este problema e aumentar a dimensão. Desta maneira, se mantém informação das bordas e permite ter uma rede com mais camadas convolucionais, ou seja, mais profunda.
Importante notar que o *stride* e tamanho do filtro não podem ser maior que a dimensão da matriz de entrada.

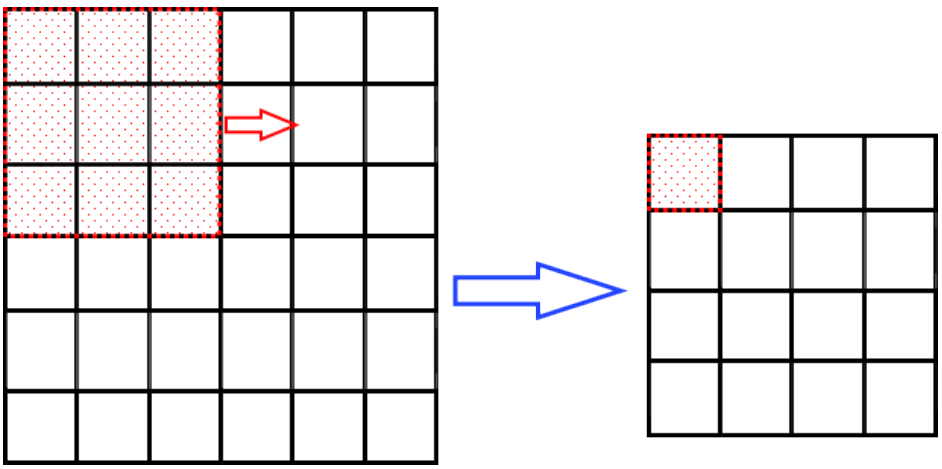


Figura 3.1: *Exemplo de filtro 3x3 com* stride *= 2 em uma imagem 6x6*

Camada de Max-Pooling

A camada de *max-pooling* permite diminui a dimensão da matriz que passa por esta camada, pois sua função é dividi-la em sub-matrizes de tamanhos menores e manter apenas o maior valor presente nesta sub-matriz. Ou seja, a matriz resultante terá os valores maiores das sub-matrizes e portanto, a matriz resultante sera composta de sub-matrizes com um único valor.
Apresenta 2 hiper-parâmetros, a extensão espacial f e *stride* s. A extensão espacial define a dimensão das sub-matrizes. Portanto, se temos $f = 2$ e $s = 2$, e uma matriz de entrada de tamanho $4x4$, a matriz resultante desta camada terá dimensão $2x2$.
A dimensão da matriz que resulta desta camada é calculada da seguinte maneira : Dado a matriz de entrada que irá passar por esta camada e w_1 sua largura e h_1 sua altura, então sua dimensão de saída w_2 e h_2 é tal :

$$w_2 = \frac{w_1 - f}{s + 1}$$

$$h_2 = \frac{h_1 - f}{s + 1}$$

Importante notar, que caso a matriz de entrada seja tridimensional, sendo a terceira dimensão a referencia a quantidade de canais, esta dimensão permanecerá e não sera removida.
Em geral, é utilizada após uma camada convolucional, porém, há um consenso entre os modeladores de usar uma camada *max-pooling* a cada 2 camadas de convolução.
Esta camada não tem pesos e , portanto, não é afetada pelo algoritmo de treinamento. Além disso, promove um tipo de invariância a translação e reduz o custo computacional para as outras camadas.

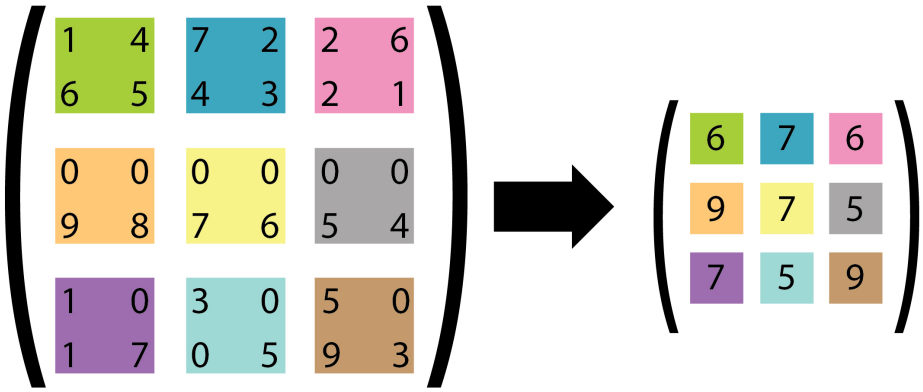


Figura 3.2: *Exemplo de max pooling com $f = 2$ e $s = 2$*

Capítulo 4

Aceleradores

4.1 Aceleração Por GPU

A aceleração por GPU é interessante pois *Deep Learning* envolve grandes quantidades de multiplicação de matrizes e convoluções, além disto, uma camada no modelo neural tem tipos idênticos de neurônios, definidos por uma função de ativação e operação matemática, portanto, fazem o mesmo tipo de computação.

No treinamento de redes neurais é utilizado pontos flutuantes pois, em geral, o conjunto de dados deve ser normalizado entre 0 e 1 ou entre -1 e 1, devido a restrição imposta pelas funções de ativações escolhidas. A melhor GPU da NVIDIA atualmente é a *NVIDIA TITAN X* a qual faz 6600 GFLOPS Walton [2015], esta quantidade é muito superior ao que uma CPU consegue fazer, por exemplo, 354 GFLOPS com uma *Core i7 5960X (Haswell E)* Kinghorn.

4.1.1 CUDA e CuDNN

Em geral, os frameworks de *Deep Learning* disponíveis atualmente, tem suporte a bibliotecas da NVIDIA, tal como o *CUDA* CUD e de tal maneira que o programador não precisa fazer a paralelização em baixo nível dos algoritmos, pois estes já vem implementados. Entre as bibliotecas que são implementadas por estes frameworks, é importante ressaltar o *CuDNN* CUD [2017]. Este contém implementações otimizadas de rotinas padrões, tal como convoluções forward e backward, pooling, normalização e funções de ativação. O uso de CUDNN permite que o treinamento seja até 3 vezes mais rápido Jcjohnson [2017]. Uma contrapartida, é que a aceleração causada pelo uso do *CuDNN* tem um preço : a não reprodutibilidade dos experimentos, as quais podem ser mitigadas pelo uso de certas flags de compilação, porém, não são eliminadas totalmente cud.

NVIDIA Deep Learning SDK

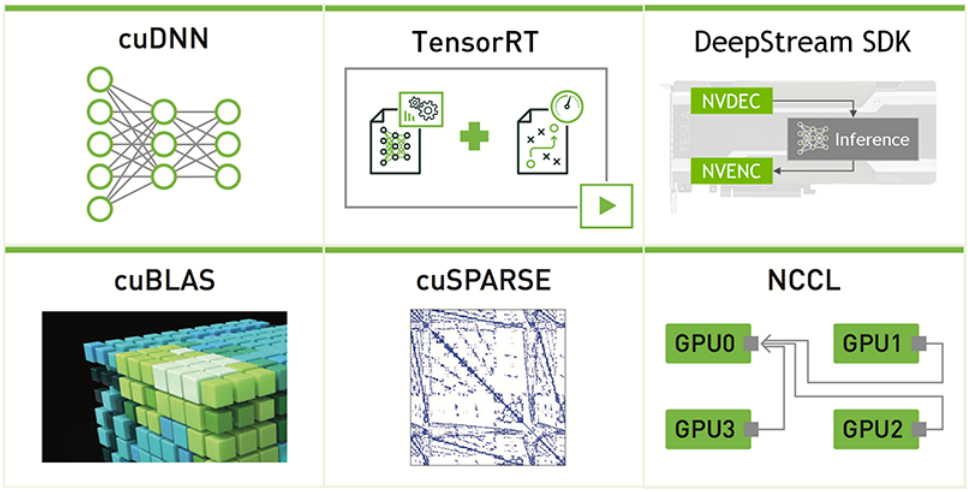


Figura 4.1: *NVIDIA Deep Learning SDK*

O *CuDNN* faz parte de um kit de desenvolvimento para *Deep Learning* da *NVIDIA*, chamada de *NVIDIA Deep Learning SDK* nvi [2017b]. Este pacote contém as bibliotecas *TensorRT* ten [2017], *NCCL* NCC [2017], *DeepStream SDK* Dee [2017], *cuBLAS* cuB [2017] e *cuSparse* cuS [2017].

O *TensorRT* é utilizado para otimizar modelos de redes neurais para inferência e pode ser utilizado para preparar o modelo antes de fazer o *deploy*, por exemplo, para o setor automotivo, o qual necessita que o processo de inferência seja rápido. *TensorRT* otimiza o processo de inferência ao diminuir a precisão de ponto flutuante de 32 bits para 16 bits ou para valores no conjunto dos inteiros, em 8 bits, e assim, reduz a latência, permitindo que em situações que demandem uso em tempo-real, não haja atraso de processamento. Esta biblioteca é gratuita e permite a acelerar até 45 vezes a tarefa de inferência e com latência inferior a 7 milissegundos. Importante ressaltar que também permite a otimização usando precisão total : ponto flutuante de 32 bits.

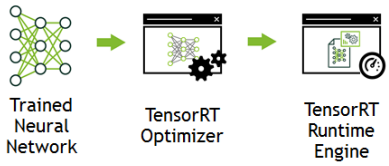


Figura 4.2: *TensorRT*

O *NCCL* permite a comunicação entre várias GPU's, ou seja, de forma que é possível executar um algoritmo de forma paralelizada em mais de uma GPU NVIDIA. *NCCL* contém primitivas otimizadas, tal como redução e broadcast . Há o suporte para GPU's instaladas em uma mesma máquina e MPI.

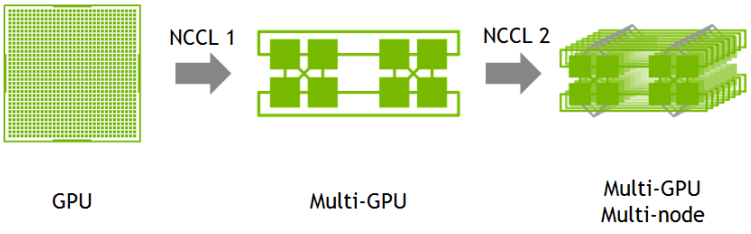


Figura 4.3: *NCCL*

DeepStream SDK permite alto desempenho ao usar redes profundas com vídeos, utilizando a linguagem C++. Esta biblioteca utiliza *TensorRT* para fazer a otimização do modelo de rede neural.

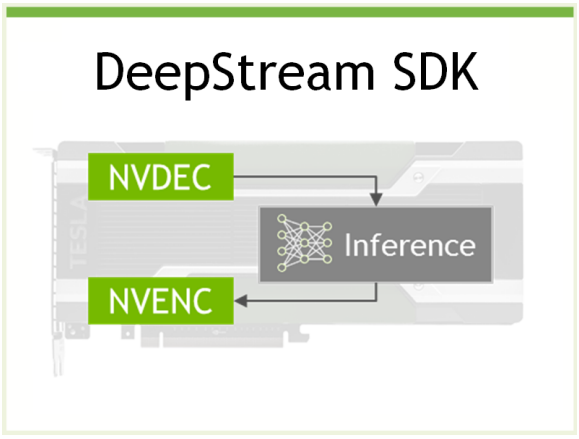


Figura 4.4: *DeepStream SDK*

cuBLAS e *cuSparse* têm como objetivo fazer cálculos algébricos e com matrizes esparsas, respectivamente. E estes têm primitivas prontas para trabalharem em GPU.

NVIDIA Deep Learning SDK é usado em diversos frameworks nvi [2017a], entre eles, *TensorFlow*, *CNTK* e *Theano*.

Theano Al-Rfou e Guillaume Alain [2016] foi desenvolvido para a linguagem Python e é mantida pelo grupo LISA da universidade de Montreal. Este permite que um mesmo código seja executado em CPU com paralelização OMP ou em GPU, e esta mudança é feita apenas mudando uma flag de execução. *Theano* permite definir, otimizar e verificar expressões matemáticas de forma eficiente.

TensorFlow Abadi e Ashish Agarwal [2015] é mantido pelo *Google* e é feito para o Python e C++ , e há um suporte em teste para java. Este é uma biblioteca de código livre para computações numéricas.

CNTK (*Microsoft Cognitive Toolkit*) Seide e Agarwal [2016] é mantido pela Microsoft e tem suporte a C++, CSHARP/.NET, Python, Java. È uma biblioteca de código livre, unificada para *Deep Learning*.

Figura 4.5: *Frameworks que suportam CUDNN*

4.1.2 DIGITS

Além do kit de desenvolvimento *NVIDIA Deep Learning SDK*, a NVIDIA disponibiliza o software *DIGITS* DIG [2017], o qual permite facilitar o desenvolvimento de redes profundas.

Usando *DIGITS* é possível criar, treinar e visualizar em tempo real as funções de perda durante a sua execução. Além disto, permite usar modelos de redes neurais disponíveis no *DIGITS*, fazer busca em grade de hiper-parâmetros para achar valores ótimos e o uso de multi-GPU.

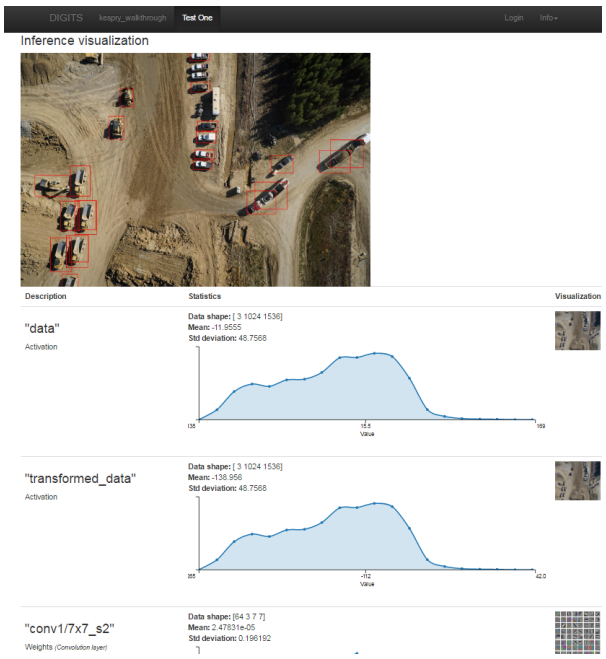


Figura 4.6: *Exemplo de inferência com DIGITS*

4.2 Aceleração por FPGA E ASIC's

A aceleração dos algoritmos de *deep learning*, no entanto, não se restringe apenas à utilização de GPU's de propósito geral. O número crescente de aplicações de *deep learning* tem motivado uma busca incessante por requisitos ótimos de desempenho e eficiência energética. Neste cenário, duas soluções parecem ter um futuro bastante promissor: os FPGA's e os ASIC's.

4.2.1 FPGA's

Os FPGA's (do inglês, Field Programmable Gate Array) são circuitos integrados reprogramáveis, compreendendo uma categoria especial de hardware reconfigurável. Isso significa, por exemplo, que é possível projetar um circuito lógico para implementar o modelo de um neurônio artificial e combiná-los para implementar uma rede neural em nível de hardware. Essa mudança de paradigma produz modelos muito mais eficientes, tanto em termos de desempenho, quanto em consumo de energia. Além disso, uma rede neural implementada em um FPGA é capaz respo

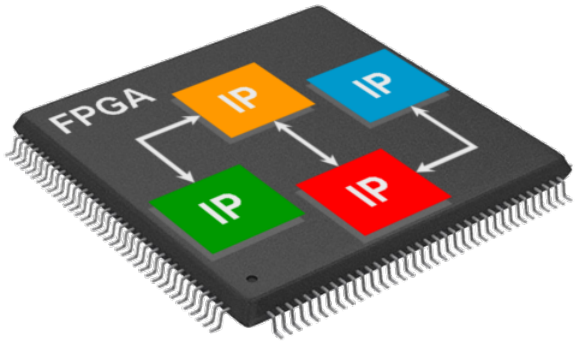


Figura 4.7: *Ilustração abstrata de uma FPGA*

Segundo [Hemsoth, 2016], as CPU's não são eficientes em questão de energia e a GPU é muito ineficiente na fase de inferência, pois em geral, as aplicações em tempo real são muito sensíveis à latência no tempo de resposta da rede. Portanto, a utilização de uma FPGA seria, em tese, a solução ideal.

No entanto, a utilização de FPGA's na aceleração dos algoritmos de *deep learning* ainda é muito restrita. Essa característica se deve, principalmente, pela dificuldade de se programar os dispositivos FPGA. O paradigma de programação para as FPGA's é totalmente diferente dos métodos convencionais de programação de software. Ao invés de fazer fazer abstrações de operações matemáticas e lógicas, a programação de uma FPGA é muito mais próxima de um projeto de hardware. Essas diferenças de paradigmas resultam em grandes dificuldades para os pesquisadores das áreas de aprendizado de máquina, que em geral estão familiarizados com o desenvolvimento de códigos de alto-nível.

Existem alguns esforços para desenvolver compiladores capazes de converter códigos escritos em alto-nível para um projeto de hardware executável em uma FPGA. No entanto, ainda não existe uma ferramenta bem consolidada específica para as aplicações de *deep learning*. Desta forma os projetos de redes neurais em FPGAs exigem muito tempo e esforço, quando comparado com as soluções em CPU's e GPU's.

As dificuldades de programação, aliadas ao alto custo e ao número restrito de equipamentos e ferramentas de desenvolvimento disponíveis no mercado, justificam o fato de que o uso de FPGA's para *deep learning* ainda é muito restrito em aplicações práticas.

4.2.2 ASIC's

O desenvolvimento de circuitos integrados específicos para aplicações de aprendizado de máquina, e, particularmente, *deep learning*, parece ser uma tendência bastante promissora para a área de inteligência artificial em um futuro próximo.

Atualmente, ainda não existe nenhum ASIC (Application Specific Integrated Circuit) específico para aplicações de *deep learning* que esteja disponível comercialmente. No entanto, existem alguns projetos bastante relevantes nesse sentido, dentre os quais devemos destacar: *i)* Google TPU, *ii)* Nervana Engine, e *iii)* Graphcore IPU.

Dentre os três projetos citados, sem dúvida o mais notável até agora é a TPU (Tensor Processing Unit), desenvolvida pelo Google e otimizado para sua própria biblioteca de aprendizado de máquina, o *Tensor Flow*. Apesar de ainda não ser um produto comercial, a TPU é o único hardware específico para *deep learning* que já está em operação (apenas nos servidores do Google).

Em um artigo publicado recentemente [Jouppi *et al.*, 2017], os desenvolvedores do Google divulgaram algumas características do hardware e os resultados obtidos. Todas as características e detalhes de arquitetura do hardware foram projetados para otimizar as operações específicas de *deep learning*, particularmente na fase de inferência. A figura abaixo representa um diagrama esquemático da estrutura interna da TPU.

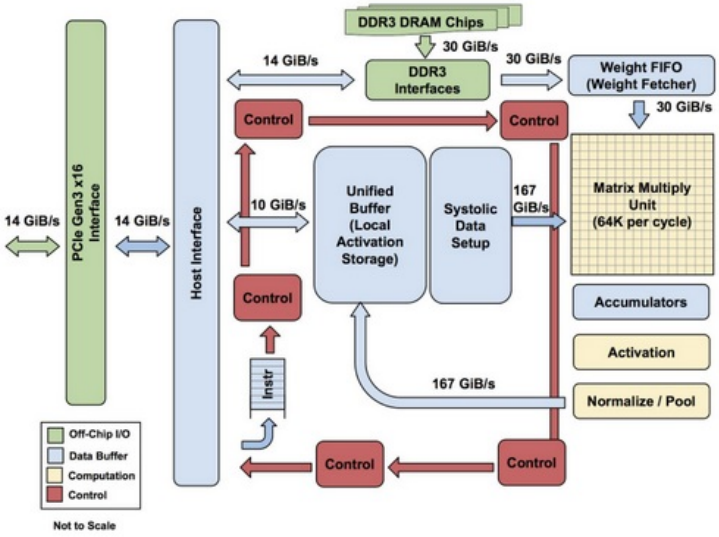


Figura 4.8: *Diagrama esquemático da TPU*

Entre as principais características da TPU, devemos destacar:

- **Unidade de multiplicação de matrizes:** projetada para realizar operações de multiplicação matricial em poucos ciclos de clock. Como a propagação dos sinais em uma rede neural é basicamente uma operação de multiplicação matricial, essa unidade é a principal novidade
- **Unidade de ativação:** uma unidade específica para aplicar funções de ativação pré-definidas e implementadas em nível de hardware, construída
- **Arquitetura de 8 bits:** O TensorFlow (biblibioteca de machine learning, para o qual a TPU foi projetada) permite utilizar uma técnica de quantização para reduzir a precisão numérica dos dados. Na fase de inferência, em geral, é possível obter uma acurácia satisfatória com uma arquitetura de 8 bits. Segundo os autores as multiplicações de 8 bits podem ser 6X mais eficientes em termos de consumo de energia e área; no caso da operação de soma, essa vantagem pode chegar a 13X em energia e 38X em área ([Dally, 2015] *apud* [Jouppi *et al.*, 2017]). A figura 4.9 ilustra como funciona a estratégia de quantização usada pelo *Tensor Flow*.

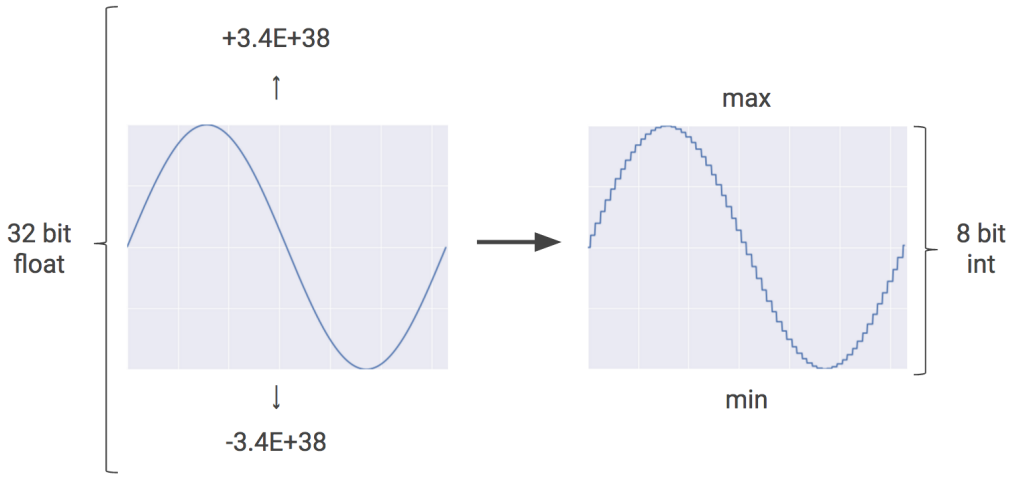


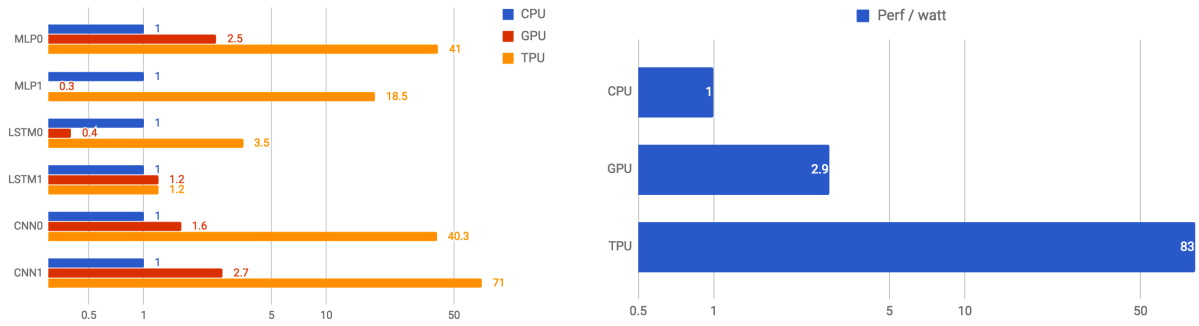
Figura 4.9: *Estratégia de quantização do Tensor Flow*

- **Instruções de hardware próprias para *deep learning*:** a TPU utiliza uma modelo CISC com instruções de hardware projetadas especificamente para aplicações de *deep learning*, conforme os exemplos da tabela 4.1

Tabela 4.1: *Instruções da TPU*

Instrução	Função
Read_Host_Memory	Carrega os dados da memória da CPU para TPU
Read_Weights	Carrega os pesos sinápticos para a memória da TPU
MatrixMultiply/Convolve	Multiplicação ou convolução entre os dados e os pesos sinápticos, acumula os resultados
Activate	Aplica uma determinada função de ativação
Write_Host_Memory	Envia os resultados para a memória da CPU

Para avaliar os ganhos proporcionados com o a utilização das TPU's, Jouppi *et al.* [2017] compararam seu desempenho em relação às CPU's e GPU's que estavam disponíveis no *datacenter* da Google (CPU Intel Haswell E5-2699 v3 [Intel, 2017]; e GPU Nvidia Tesla K80 [NVIDIA, 2017]). A figura A figura 4.10 demonstra um resultado impressionante, tanto em termos de desempenho, quanto em eficiência energética. A figura 4.10 Interessante notar que o desempenho da CPU foi



(a) *Desempenho (inferências/segundo)* **(b)** *Eficiência energética*

Figura 4.10: *Resultados divulgados pelo Google nos testes com a TPU [Sato et al., 2017].*

normalizado para 1, e o eixo horizontal foi comprimido para facilitar a visualização. O ganho em eficiência energética chega a ser superior à 80X e, nos casos mais extremos, o desempenho de inferência chegou a superar 70X.

Além da TPU, outras duas iniciativas neste sentido também parecem bastante promissoras: o Nervana Engine e a GraphCore IPU.

A Nervana, foi uma startup fundada em 2014 com o objetivo de desenvolver hardwares otimizados para aplicações de aprendizado de máquina e que demandam alta carga computacional. Em 2016, a Intel, o maior fabricante mundial de microprocessadores para computadores pessoais, adquiriu a Nervana por mais de 400 milhões de dólares, fato que ressalta a importância deste nicho de mercado em um futuro próximo. A promessa da Intel é que o *Nervana Engine* seja lançado ainda em 2017. Os fabricantes ainda não divulgaram muitos detalhes sobre o hardware nem sobre o ferramental de desenvolvimento para estes equipamentos. No entanto, algumas informações oficiais já foram confirmadas [Kloss, 2016]:

- O sistema completo será composto por 8 ASIC's conectados em uma configuração *Torus*
- Cada ASIC contará com 4Gb de memória, totalizando 32 Gb
- Arquitetura de 16 bits
- A velocidade de acesso à memória pode alcançar até 8 terabits/s
- A tecnologia de fabricação 28nm, com promessa de redução para até 16nm

Neste mesmo segmento de mercado, uma outra empresa também promete entregar um produto comercial ainda em 2017, a GraphCore. A empresa recebeu aportes de diversos fundos de investimentos, dentre os quais estão a Samsung e a Dell. Seu principal produto, chamado IPU (Intelligence Processing Unit), promete acelerar os algoritmos de *deep learning* tanto na fase de treinamento, quanto de inferência [Toon, 2017]. Os detalhes técnicos ainda não foram revelados, no entanto, entre algumas informações divulgadas no próprio site da empresa, destacam-se:

- O ganho de desempenho estimado deve ser da ordem de 10x a 100x em comparação aos melhores sistemas atuais
- Suporte à um *framework* próprio, chamado Poplar [Flyes, 2017] (semelhante ao CUDA para GPU's NVIDIA), com interface para as linguagens C++ e Python
- Desenvolvimento de uma linha de IPU's de baixa potência, própria para sistemas embarcados

A figura 4.11, divulgada no site do fabricante, indica que haverá suporte às principais bibliotecas de aprendizado de máquina disponíveis atualmente. A figura também ilustra a papel do Poplar, uma ferramenta de desenvolvimento própria que irá fazer a interface entre os algoritmos as bibliotecas e os recursos de hardware.

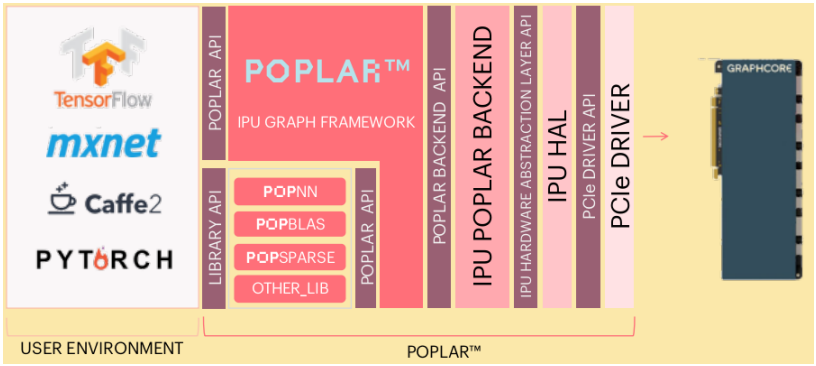


Figura 4.11: *GraphCore IPU e Poplar*

.

Capítulo 5

Conclusão

Em suma, o campo de aplicação de rede neurais profundas tem expandido consideravelmente, e estas exigem altíssima capacidade computacional. Atualmente, há pesquisas que consideram sistemas embarcados usando otimizadores, como por exemplo, *TensorRT* e portanto a otimização destes modelos e a disponibilidade de tecnologia para o aceleramento deles, é muito importante. Pode-se citar um caso específico : carros autônomos Huval *et al.* [2015]. Neste caso, há a necessidade de um equipamento embarcado que tenha pouca latência. Porém, em geral, não é desejável que o usuário perca tempo esperando o processamento.

Vale ressaltar que há outras áreas interessantes que estão iniciando o uso de aprendizado profundo, entre elas, um projeto de aprendizado por reforço para jogos : *Open AI Universe* ope, a qual permite o uso de modelos de *Deep Learning* Futurely [2017]

Apesar do avanço de *FPGA's* e outros hardwares específicos, a utilização de GPU's ainda é a melhor solução para o treinamento destes modelos devido ao desempenho que estes dispositivos oferecem em relação aos outros. Um dos motivos desta diferença, é o fato de a *NVIDIA* ter investido bastante na biblioteca *CUDA* e no *NVIDIA Deep Learning SDK* e devido a isto, estas bibliotecas além de apresentarem um ótimo desempenho, também tem uma ótima qualidade de código. Em contrapartida, a biblioteca equivalente de código aberto, *OpenCL*, em geral apresenta um desempenho inferior e há uma menor quantidade de bibliotecas de *Deep learning* que oferecem suporte a ela. No *Theano* e *Tensorflow* por exemplo, o suporte a *OpenCL* ainda é incompleto the, Tensorflow.

Há diversas empresas, por exemplo a *Intel* int e a *DeePhi*, estudando outras maneiras de realizar esta tarefa, de forma a ter um aumento de eficiência energética e desempenho, usando por exemplo, *FPGA's*.

Referências Bibliográficas

CUD() Plataforma de computação paralela | cuda | nvidia | nvidia. URL http://www.nvidia.com.br/object/cuda_home_new_br.html. Citado na pág. 10

CUD(2017) Nvidia cudnn, Jun 2017. URL https://developer.nvidia.com/cudnn. Citado na pág. 10

DIG(2017) Digits, Jun 2017. URL https://developer.nvidia.com/digits. Citado na pág. 12

Dee(2017) Nvidia deepstream sdk, Apr 2017. URL https://developer.nvidia.com/deepstream-sdk. Citado na pág. 11

NCC(2017) Nccl, Jun 2017. URL https://developer.nvidia.com/NCCL. Citado na pág. 11

cuB(2017) cublas, Jun 2017. URL https://developer.nvidia.com/cublas. Citado na pág. 11

cuS(2017) cusparse, Jun 2017. URL https://developer.nvidia.com/cuSPARSE. Citado na pág. 11

cud() theano.sandbox.cuda.dnn – cudnn¶. URL http://deeplearning.net/software/theano/library/sandbox/cuda/dnn.html. Citado na pág. 10

int() Intel® deep learning inference accelerator-image recognition. URL https://www.intel.com/content/www/us/en/servers/accelerators/deep-learning-inference-accelerator-product-detail.html. Citado na pág. 18

nvi(2017a) Deep learning frameworks, Apr 2017a. URL https://developer.nvidia.com/deep-learning-frameworks. Citado na pág. 12

nvi(2017b) Deep learning software, May 2017b. URL https://developer.nvidia.com/deep-learning-software. Citado na pág. 11

ope() Universe. URL https://universe.openai.com/. Citado na pág. 18

ten(2017) Nvidia tensorrt, Jun 2017. URL https://developer.nvidia.com/tensorrt. Citado na pág. 11

the() Using the gpu¶. URL http://deeplearning.net/software/theano/tutorial/using_gpu.html. Citado na pág. 18

Abadi e Ashish Agarwal(2015) Martín Abadi e et al. Ashish Agarwal. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org. Citado na pág. 12

Al-Rfou e Guillaume Alain(2016) Rami Al-Rfou e et al. Guillaume Alain. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688. URL http://arxiv.org/abs/1605.02688. Citado na pág. 12

Csáji(2001) Balázs Csanád Csáji. Approximation with artificial neural networks. *MSc Thesis, Eötvös Loránd University (ELTE), Budapest, Hungary.* Citado na pág. 5

Dally(2015) William Dally. High-performance hardware for machine learning. *NIPS Tutorial.* Citado na pág. 15

F.() Rosenblatt F. The perceptron, a perceiving and recognizing automaton <project para>. Citado na pág. 1

Flyes(2017) Matt Flyes. Graph computing for machine intelligence with poplar, 2017. URL https://www.graphcore.ai/posts/graph-computing-for-machine-intelligence-with-poplar. Official GraphCore Blog. Citado na pág. 16

Fukushima(1980) Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202. ISSN 1432-0770. doi: 10.1007/BF00344251. URL http://dx.doi.org/10.1007/BF00344251. Citado na pág. 7

Futurely(2017) Futurely. futurely/openai-universe-agents, Feb 2017. URL https://github.com/futurely/openai-universe-agents. Citado na pág. 18

Hemsoth(2016) Nicole Hemsoth. Fpga based deep learning accelerators take on asics, 2016. URL https://www.nextplatform.com/2016/08/23/fpga-based-deep-learning-accelerators-take-asics/. Citado na pág. 13

Hinton *et al.***(2012)** Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever e Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580. URL http://arxiv.org/abs/1207.0580. Citado na pág. 7

Hornik(1991) Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(91)90009-T. URL http://www.sciencedirect.com/science/article/pii/089360809190009T. Citado na pág. 5

Huval *et al.***(2015)** Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue *et al.* An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*. Citado na pág. 18

Intel(2017) Intel. Processador intel® xeon® e5-2699 v3, 2017. URL https://ark.intel.com/pt-br/products/81061/Intel-Xeon-Processor-E5-2699-v3-45M-Cache-2_30-GHz. Citado na pág. 15

Jcjohnson(2017) Jcjohnson. jcjohnson/cnn-benchmarks, Jun 2017. URL https://github.com/jcjohnson/cnn-benchmarks. Citado na pág. 10

Jouppi *et al.***(2017)** Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers *et al.* In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760*. Citado na pág. 14, 15

Kinghorn() Dr Donald Kinghorn. Linpack performance haswell e (core i7 5960x and 5930k). URL https://www.pugetsystems.com/labs/hpc/Linpack-performance-Haswell-E-Core-i7-5960X-and-5930K-594/. Citado na pág. 10

Kloss(2016) Carey Kloss. Nervana engine delivers deep learning at ludicrous speed!, 2016. URL https://www.intelnervana.com/nervana-engine-delivers-deep-learning-at-ludicrous-speed/. Official Nervana Blog. Citado na pág. 16

LeCun *et al.***(2001)** Y. LeCun, L. Bottou, Y. Bengio e P. Haffner. Gradient-based learning applied to document recognition. Em *Intelligent Signal Processing*, páginas 306–351. IEEE Press. Citado na pág. 7

Lecun e Bengio(1995) Yann Lecun e Yoshua Bengio. *Convolutional Networks for Images, Speech and Time Series*, páginas 255–258. The MIT Press. Citado na pág. 7

McCulloch e Pitts(1943) Warren S. McCulloch e Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133. doi: 10.1007/BF02478259. URL http://dx.doi.org/10.1007/BF02478259. Citado na pág. 3, 7

Mitchell(1997) Tom Mitchell. *Machine Learning*. McGraw-Hill Education, 1º edição. Tradução Nossa. Citado na pág. 2

NVIDIA(2017) NVIDIA. Nvidia tesla k80, 2017. URL http://www.nvidia.com/object/tesla-k80.html. Citado na pág. 15

Nvidia(2010) CUDA Nvidia. Programming guide, 2010. Citado na pág. 1

Sato *et al.***(2017)** Kaz Sato, Cliff Young e David Patterson. An in-depth look at google's first tensor processing unit (tpu), 2017. URL https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu. Official Google Cloud Blog. Citado na pág. 16

Seide e Agarwal(2016) Frank Seide e Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. Em *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, páginas 2135–2135, New York, NY, USA. ACM. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2945397. URL http://doi.acm.org/10.1145/2939672.2945397. Citado na pág. 12

Tensorflow() Tensorflow. Opencl support on tensorflow. URL https://github.com/tensorflow/tensorflow/issues/22. Citado na pág. 18

Toon(2017) Nigel Toon. Machine learning processors for both training and inference, 2017. URL https://www.graphcore.ai/posts/machine-learning-processors-for-both-training-and-inference. Official GraphCore Blog. Citado na pág. 16

Walton(2015) Steven Walton. Nvidia geforce gtx titan x review, Mar 2015. URL https://www.techspot.com/review/977-nvidia-geforce-gtx-titan-x/. Citado na pág. 10