Buffer perfeito para treinamento de redes neurais

Nome: Vitor Rossi Speranza nº Usp: 10262523
Nome: Miguel de Mattos Gardini nº Usp: 10295728
Nome: Bruno Del Monde nº Usp: 10262818

Resumo

A proposta do trabalho é implementar e testar uma implementação de buffer perfeito, tendo como aplicação alvo o treinamento de redes neurais, situação onde, mesmo na vida real, com pouca preparação especial, pode-se saber quais serão as imagens acessadas no futuro, permitindo assim a aplicação do buffer perfeito.
Tal buffer pode acelerar o treinamento de uma rede neural que depende de uma base de dados muito grande e não consegue coloca-lá inteira na memória ram em um momento qualquer.

Introdução

O trabalho do grupo possui caráter exploratório e experimental. Portanto, partimos de uma hipótese de que o treinamento de redes neurais é um ambiente possível de prever o acesso a memória indefinidamente na medida em este que é definido por geração pseudo-aleatória determinística após conhecida a semente e sem interação com o usuário. Com esse conhecimento seria teoricamente possível criar um buffer sempre carregado com o que vai ser utilizado em seguida, um buffer sem faltas, ou seja, que não recebe uma requisição para a qual já não esteja preparado.
Nosso trabalho consistiu em testar a viabilidade de utilizar tal buffer no caso citado de treinamento de uma rede neural, como em também testar seu desempenho e o ganho em eficiência que ele traria para o treinamento.

Problema

O problema proposto é aplicar um buffer perfeito para melhorar o tempo de treinamento de uma rede neural. No contexto de treinamento de redes neurais, pode-se calcular com antecedência todos os acessos que serão feitos a base de

dados, podendo usar essa informação para levar ao máximo o número de colisões no buffer, evitando acessos desnecessários ao disco, e possivelmente a memória RAM caso esse algoritmo também seja aplicado a memória cache do sistema.

Abordagem

O problema foi resolvido armazenando préviamente todos os acessos que o treinamento de uma rede neural irá fazer e então usando essa informação para calcular a ordem em que os dados serão colocados ou removidos do buffer, maximizando assim o número de colisões e minimizando o tempo necessário para treinar a rede neural.

Desenvolvimento

Para realizar esse projeto, 3 problemas distintos precisaram ser resolvidos.
Primeiramente, uma rede neural funcional precisou ser criada, para isso usamos a linguagem de programação Python e a biblioteca Tensor Flow. Essas ferramentas permitiram a criação de uma rede neural simples com pouco esforço e podemos então prosseguir para o próximo problema, carregar uma base de dados para fazer o treinamento de tal rede.
A base de dados escolhida foi o MNIST, uma base com 60.000 números feitos a mão, comumente usada para avaliação de métodos de processamento de imagem. Essa database foi escolhida para esse projeto pois é aberta ao público, mas infelizmente não é grande o suficiente para encher a memória RAM de um computador comum, então partimos ao próximo problema: simular uma RAM pequena.
Para melhorar a velocidade de execução do projeto, as imagens ficam armazenadas todas em RAM e o buffer é apenas visto como um conceito, são contadas as colisões e as falhas no buffer e então é mostrada a eficiência do buffer.
Além disso, ler as imagens do MNIST se provou um desafio por conta de seu formato compactado específico, foi necessário criar um arquivo inteiro para ler e descompactar as imagens do MNIST de forma que a rede neural consiga entender.

Com esses problemas resolvidos, partimos a implementação do buffer em si.
O algoritmo de buffer perfeito é razoavelmente complexo e o livro de referência não explica que estrutura de dados usar ou como aplicá-las de forma a ter um tempo de execução razoável, por conta disso, nossa primeira tentativa de implementação do buffer tinha tempo de execução O(n) para cada acesso, o que é completamente

inviável e faz a execução do algoritmo de buffer levar mais tempo que um acesso a disco, dependendo do tamanho do buffer. Uma vez que esse resultado foi observado, passamos a pensar sobre como implementar o mesmo algoritmo de buffer perfeito com um tempo de execução razoável.
Usando a estrutura de dados dicionário, se tornou possível reduzir o tempo de acesso ao buffer de O(n) para O(log(n)), o que transformou esse algoritmo em algo usável na vida real.

Seguem alguns testes feitos com o buffer e seu desempenho.

Base de dados: 10.000 imagens para treinamento de uma rede neural
Buffer contendo no máximo 10000 imagens ou 100% da base de dados, após 100.000 consultas, 10.00% não estavam presentes no buffer, pois o buffer começa vazio
Buffer contendo no máximo 5000 imagens ou 50% da base de dados, após 100.000 consultas, 23.72% não estavam presentes no buffer.

Buffer contendo no máximo 1000 imagens ou 10% da base de dados, após 100.000 consultas, 60.14% não estavam presentes no buffer.
Buffer contendo no máximo 500 imagens ou 5% da base de dados, após 100.000 consultas, 71.02% não estavam presentes no buffer.

Buffer contendo no máximo 100 imagens ou 1% da base de dados, após 100.000 consultas, 86.88% não estavam presentes no buffer.

Buffer contendo no máximo 10 imagens ou 0.1% da base de dados, após 100.000 consultas, 96.50% não estavam presentes no buffer.

Buffer contendo no máximo 2 imagens ou 0.02% da base de dados, após 100.000 consultas, 99.23% não estavam presentes no buffer.

Buffer contendo no máximo 1 imagem ou 0.01% da base de dados, após 100.000 consultas, 99.99% não estavam presentes no buffer.

Base de dados aumentada para todas as 60.000 imagens do mnist para treinamento de uma rede neural.

Buffer contendo no máximo 10000 imagens ou 16.667% da base de dados, após 100.000 consultas, 57.78% não estavam presentes no buffer.

Buffer contendo no máximo 5000 imagens ou 8.333% da base de dados, após 100.000 consultas, 66.77% não estavam presentes no buffer.

Buffer contendo no máximo 1000 imagens ou 1.667% da base de dados, após 100.000 consultas, 83.44% não estavam presentes no buffer.

Buffer contendo no máximo 500 imagens ou 0.833% da base de dados, após 100.000 consultas, 88.02% não estavam presentes no buffer.

Buffer contendo no máximo 100 imagens ou 0.167% da base de dados, após 100.000 consultas, 94.64% não estavam presentes no buffer.

Buffer contendo no máximo 10 imagens ou 0.016% da base de dados, após 100.000 consultas, 98.57% não estavam presentes no buffer.

Buffer contendo no máximo 2 imagens ou 0.003% da base de dados, após 100.000 consultas, 99.67% não estavam presentes no buffer.

Buffer contendo no máximo 1 imagem ou 0.0017% da base de dados, após 100.000 consultas, 99.99% não estavam presentes no buffer.

O relatório completo está no arquivo report.txt

Conclusão

Os desafios mais relevantes para esse projeto foram, além implementar uma rede neural funcional, entender e processar uma base de dados real, implementar um algoritmo de buffer, que o livro diz não ser utilizável em situações reais, em uma situação real, obtendo resultados satisfatórios ao ir um passo além do que está descrito e deduzir como implementar o algoritmo em tempo hábil.

A implementação feita é uma prova de conceito, podendo ser facilmente extrapolada para lidar com situações reais.

Esse projeto prova como fato que o algoritmo de buffer perfeito tem sim aplicações na vida real e pode aumentar significativamente a velocidade de treinamento de redes neurais ou de qualquer situação em que seja possível deduzir com antecedência quais acessos serão feitos a memória.

O mesmo algoritmo pode ser aplicado também a memória cache, aumentando ainda mais a velocidade de execução de tudo que o usar.

Referências

Sistemas Operacionais Modernos - 4º Edição - ANDREW S., TANENBAUM HERBERT.
The MNIST database of handwritten digits.
Tensor Flow, estimators guide.

Anexo 1:

Para executar esse código é necessário ter python 3.6 com os módulos numpy e tensorflow. Os comandos para instalar os módulos são os seguintes:
`pip install tensorflow`
e
`pip install numpy`

Para executar o código basta descompactar o arquivo zip em uma pasta e executar o arquivo NeuralNetwork.py usando python 3 com os módulos numpy e tensor flow, este deve treinar a rede implementada usando o buffer e então imprimir resultados como os presentes ao final do arquivo report.txt

Para instalar os módulos tensorflow e numpy basta usar os comandos:

O número de execuções de treino e o tamanho dos minibatchs pode ser modificado no começo do arquivo NeyralNetwork.py
O arquivo mnistbuffer.py contem o buffer e suas especificações, que lá podem ser alteradas, entre elas o tamanho do grupo para treinamento, o número de imagens que cabem dentro do buffer e o número total de acessos que será feito ao buffer.