|  |
| --- |
| Изображение выглядит как зарисовка, рисунок, символ, корона  Контент, сгенерированный ИИ, может содержать ошибки. |
| МИНОБРНАУКИ РОССИИ  Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** |

Институт комплексной безопасности и цифровых технологий

Кафедра КБ-14 «Цифровые технологии обработки данных»

**Платформы анализа больших данных**

**Лабораторная работа 4**

Вариант 11

Выполнил:

Студент группы БСБО-09-22

Шутов Кирилл Сергеевич

Проверил:

Кашкин Евгений Владимирович

**Москва, 2025**

Постановка задачи

Целью данной работы – изучить и сравнить различные реализации простой двухслойной нейронной сети с архитектурой 128 → 32 → 1, используя разные подходы и технологии: низкоуровневый Python (NumPy), PyTorch на CPU и GPU, C++ без использования библиотек и CUDA с применением тензорных ядер.

Описание кода и выполненных действий

Для каждого варианта реализации (см листинг 1-4) была проведена серия замеров времени выполнения полного цикла вычислений (forward + backward) с одним и тем же размером батча (8 образцов). Перед началом измерений производилось несколько "разогревающих" итераций для стабилизации производительности (кэширование данных, инициализация GPU и т.д.).

Замеры проводились следующим образом:

* для Python (NumPy) и C++ использовался стандартный таймер (time.perf\_counter(), std::chrono);
* для PyTorch и CUDA (GPU) использовались функции синхронизации и замера времени CUDA (torch.cuda.synchronize() и cudaEvent).

В таблице 1 представлены результаты замеров, время в миллисекундах на итерацию.

Таблица 1. Результаты замеров

|  |  |
| --- | --- |
| **Реализация** | **Среднее время, ms** |
| Низкоуровневый Python (NumPy) | ~12.6 |
| PyTorch CPU | ~4.0 |
| PyTorch GPU | ~0.83 |
| C++ без библиотек | ~7.5 |
| CUDA + cuBLAS (Tensor Cores) | ~0.59 |

Реализация на чистом Python (NumPy) показала наибольшее время выполнения из-за накладных расходов интерпретатора и менее оптимизированного использования аппаратных возможностей CPU.

PyTorch на CPU продемонстрировал лучшее время среди CPU-реализаций благодаря оптимизации многопоточности и эффективному использованию BLAS.

GPU-реализации значительно превзошли CPU по производительности: PyTorch на GPU оказался примерно в 5 раз быстрее, чем CPU-реализация, благодаря параллелизму вычислений.

CUDA-реализация с использованием cuBLAS и тензорных ядер оказалась наиболее быстрой, примерно в 1.4 раза быстрее PyTorch GPU, за счет более точного контроля и использования специализированных функций с минимальными накладными расходами.

Вывод

В рамках выполнения практической работы была продемонстрирована значительная разница в производительности различных подходов к реализации простой нейронной сети. GPU-реализации, особенно с использованием низкоуровневых технологий (CUDA и тензорные ядра), предоставляют существенный выигрыш по скорости, который становится еще более заметным при увеличении размера данных и сети. Однако использование готовых библиотек, таких как PyTorch, позволяет существенно упростить процесс разработки и отладки с небольшими компромиссами по скорости.

Источники

1. Документация NVIDIA CUDA. [Электронный ресурс] URL: https://docs.nvidia.com/cuda/ Дата обращения: (03.03.2025 г).
2. Shared Memory Optimizations. [Электронный ресурс] URL: https://docs.nvidia.com/cuda/ Дата обращения: (03.04.2025 г).

**Листинг**

|  |
| --- |
| import time import numpy as np  in\_dim = 128 hidden\_dim = 32 out\_dim = 1 batch\_size = 8  np.random.seed(0) W1 = np.random.randn(in\_dim, hidden\_dim).astype(np.float32) b1 = np.zeros(hidden\_dim, dtype=np.float32) W2 = np.random.randn(hidden\_dim, out\_dim).astype(np.float32) b2 = np.zeros(out\_dim, dtype=np.float32)   def forward\_backward(X, Y\_true):  *# Forward* Z1 = X.dot(W1) + b1 *# (batch\_size, hidden\_dim)* H = np.maximum(Z1, 0) *# ReLU* Y\_pred = H.dot(W2) + b2 *# (batch\_size, out\_dim)   # Loss и градиент по выходу* dY = (Y\_pred - Y\_true) *# MSE derivative (без 1/2)   # Backward* dW2 = H.T.dot(dY) *# (hidden\_dim, out\_dim)* db2 = dY.sum(axis=0)  dH = dY.dot(W2.T) *# (batch\_size, hidden\_dim)* dZ1 = dH \* (Z1 > 0) *# ReLU'* dW1 = X.T.dot(dZ1) *# (in\_dim, hidden\_dim)* db1 = dZ1.sum(axis=0)   return dW1, db1, dW2, db2   X = np.random.randn(batch\_size, in\_dim).astype(np.float32) Y = np.random.randn(batch\_size, out\_dim).astype(np.float32)  for \_ in range(2):  forward\_backward(X, Y)  times = [] for i in range(10):  t0 = time.perf\_counter()  grads = forward\_backward(X, Y)  t1 = time.perf\_counter()  times.append((t1 - t0) \* 1000)print("Низкоуровневый Python (NumPy), ms per iter:") print(times) |

Листинг 1. nn\_numpy.py

|  |
| --- |
| import argparse import time import torch import torch.nn as nn  parser = argparse.ArgumentParser() parser.add\_argument('--device', type=str, default='cpu', choices=['cpu', 'cuda']) args = parser.parse\_args()  device = torch.device(args.device)  in\_dim, hidden\_dim, out\_dim = 128, 32, 1 batch\_size = 8  model = nn.Sequential(  nn.Linear(in\_dim, hidden\_dim),  nn.ReLU(),  nn.Linear(hidden\_dim, out\_dim) ).to(device)  criterion = nn.MSELoss() optimizer = torch.optim.SGD(model.parameters(), lr=0.01)  X = torch.randn(batch\_size, in\_dim, device=device) Y = torch.randn(batch\_size, out\_dim, device=device)  for \_ in range(2):  optimizer.zero\_grad()  pred = model(X)  loss = criterion(pred, Y)  loss.backward()  optimizer.step()  times = [] for i in range(10):  torch.cuda.synchronize() if device.type == 'cuda' else None  t0 = time.perf\_counter()  optimizer.zero\_grad()  pred = model(X)  loss = criterion(pred, Y)  loss.backward()  optimizer.step()  torch.cuda.synchronize() if device.type == 'cuda' else None  t1 = time.perf\_counter()  times.append((t1 - t0) \* 1000)  print(f"PyTorch ({device.type.upper()}) ms per iter:") print(times) |

Листинг 2. nn\_pytorch.py

|  |
| --- |
| #include <vector> #include <iostream> #include <random> #include <chrono> using namespace std; using Clock = chrono::high\_resolution\_clock;  const int IN\_DIM = 128; const int HIDDEN\_DIM = 32; const int OUT\_DIM = 1; const int BATCH = 8;  inline float relu(float x) { return x > 0 ? x : 0; } inline float relu\_deriv(float x) { return x > 0 ? 1 : 0; }  int main() { mt19937 gen(0);  normal\_distribution<float> dist(0.0f, 1.0f);  vector<float> W1(IN\_DIM \* HIDDEN\_DIM), b1(HIDDEN\_DIM);  vector<float> W2(HIDDEN\_DIM \* OUT\_DIM), b2(OUT\_DIM);    for (auto &w : W1) w = dist(gen);  for (auto &w : W2) w = dist(gen);  vector<float> X(BATCH \* IN\_DIM), Y(BATCH \* OUT\_DIM);  for (auto &x : X) x = dist(gen);  for (auto &y : Y) y = dist(gen);  vector<float> Z1(BATCH \* HIDDEN\_DIM), H(BATCH \* HIDDEN\_DIM);  vector<float> Y\_pred(BATCH \* OUT\_DIM);  vector<float> dW1(IN\_DIM \* HIDDEN\_DIM), db1(HIDDEN\_DIM);  vector<float> dW2(HIDDEN\_DIM \* OUT\_DIM), db2(OUT\_DIM);  auto run\_once = [&]() {  *// Forward* for (int n=0; n<BATCH; ++n) {  for (int j=0; j<HIDDEN\_DIM; ++j) {  float sum = b1[j];  for (int i=0; i<IN\_DIM; ++i)  sum += X[n\*IN\_DIM + i] \* W1[i\*HIDDEN\_DIM + j];  Z1[n\*HIDDEN\_DIM + j] = sum;  H[n\*HIDDEN\_DIM + j] = relu(sum);  }  for (int k=0; k<OUT\_DIM; ++k) {  float sum = b2[k];  for (int j=0; j<HIDDEN\_DIM; ++j)  sum += H[n\*HIDDEN\_DIM + j] \* W2[j\*OUT\_DIM + k];  Y\_pred[n\*OUT\_DIM + k] = sum;  }  }  *// Backward (MSE loss)* for (int n=0; n<BATCH; ++n) {  for (int k=0; k<OUT\_DIM; ++k) {  float dy = (Y\_pred[n\*OUT\_DIM + k] - Y[n\*OUT\_DIM + k]);  db2[k] += dy;  for (int j=0; j<HIDDEN\_DIM; ++j)  dW2[j\*OUT\_DIM + k] += H[n\*HIDDEN\_DIM + j] \* dy;  for (int j=0; j<HIDDEN\_DIM; ++j) {  float dz = W2[j\*OUT\_DIM + k] \* dy \* relu\_deriv(Z1[n\*HIDDEN\_DIM + j]);  db1[j] += dz;  for (int i=0; i<IN\_DIM; ++i)  dW1[i\*HIDDEN\_DIM + j] += X[n\*IN\_DIM + i] \* dz;  }  }  }  };  run\_once();  run\_once();  vector<double> times;  for (int it=0; it<10; ++it) { fill(dW1.begin(), dW1.end(), 0);  fill(db1.begin(), db1.end(), 0);  fill(dW2.begin(), dW2.end(), 0);  fill(db2.begin(), db2.end(), 0);   auto t0 = Clock::now();  run\_once();  auto t1 = Clock::now();  times.push\_back(chrono::duration<double, milli>(t1 - t0).count());  }   cout << "C++ no-libs ms per iter: ";  for (auto t : times) cout << t << ' ';  cout << ' ';  return 0; } |

Листинг 3. nn.cpp

|  |
| --- |
| #include <cuda.h> #include <cuda\_runtime.h> #include <cublas\_v2.h> #include <iostream> #include <vector> #include <random>  const int IN\_DIM = *128*; const int HIDDEN\_DIM = *32*; const int OUT\_DIM = *1*; const int BATCH = *8*; #define CUDA\_CHECK(err) if(err!=cudaSuccess){std::cerr<<cudaGetErrorString(err);return -*1*;} #define CUBLAS\_CHECK(err) if(err!=CUBLAS\_STATUS\_SUCCESS){std::cerr<<"cuBLAS error";return -*1*;}  \_\_global\_\_ void add\_bias\_relu(float\* Z, const float\* b, int batch, int dim) {  int idx = blockIdx.x \* blockDim.x + threadIdx.x;  if (idx < batch\*dim) {  int j = idx % dim;  float v = Z[idx] + b[j];  Z[idx] = (v > *0* ? v : *0*);  } }  int main() { std::mt19937 gen(*0*);  std::normal\_distribution<float> dist(*0*,*1*);  std::vector<float> h\_X(BATCH\*IN\_DIM), h\_Y(BATCH\*OUT\_DIM);  std::vector<float> h\_W1(IN\_DIM\*HIDDEN\_DIM), h\_b1(HIDDEN\_DIM);  std::vector<float> h\_W2(HIDDEN\_DIM\*OUT\_DIM), h\_b2(OUT\_DIM);  for (auto& x : h\_X) x = dist(gen);  for (auto& y : h\_Y) y = dist(gen);  for (auto& w : h\_W1) w = dist(gen);  for (auto& w : h\_W2) w = dist(gen);   *cublasHandle\_t* handle;  CUBLAS\_CHECK(cublasCreate(&handle));  cublasSetMathMode(handle, CUBLAS\_TENSOR\_OP\_MATH);  float \*d\_X, \*d\_Z1, \*d\_H, \*d\_Ypred;  float \*d\_W1, \*d\_b1, \*d\_W2, \*d\_b2;  CUDA\_CHECK(cudaMalloc(&d\_X, BATCH\*IN\_DIM\*sizeof(float)));  CUDA\_CHECK(cudaMalloc(&d\_Z1, BATCH\*HIDDEN\_DIM\*sizeof(float)));  CUDA\_CHECK(cudaMalloc(&d\_Ypred, BATCH\*OUT\_DIM\*sizeof(float)));  CUDA\_CHECK(cudaMalloc(&d\_W1, IN\_DIM\*HIDDEN\_DIM\*sizeof(float)));  CUDA\_CHECK(cudaMalloc(&d\_b1, HIDDEN\_DIM\*sizeof(float)));  CUDA\_CHECK(cudaMalloc(&d\_W2, HIDDEN\_DIM\*OUT\_DIM\*sizeof(float)));  CUDA\_CHECK(cudaMalloc(&d\_b2, OUT\_DIM\*sizeof(float)));   CUBLAS\_CHECK(cublasSetVector(BATCH\*IN\_DIM, sizeof(float), h\_X.data(), *1*, d\_X, *1*));  CUBLAS\_CHECK(cublasSetVector(BATCH\*OUT\_DIM, sizeof(float), h\_Y.data(), *1*, d\_Ypred, *1*)); *// reuse for Y* CUBLAS\_CHECK(cublasSetVector(IN\_DIM\*HIDDEN\_DIM, sizeof(float), h\_W1.data(), *1*, d\_W1, *1*));  CUBLAS\_CHECK(cublasSetVector(HIDDEN\_DIM, sizeof(float), h\_b1.data(), *1*, d\_b1, *1*));  CUBLAS\_CHECK(cublasSetVector(HIDDEN\_DIM\*OUT\_DIM, sizeof(float), h\_W2.data(), *1*, d\_W2, *1*));  CUBLAS\_CHECK(cublasSetVector(OUT\_DIM, sizeof(float), h\_b2.data(), *1*, d\_b2, *1*));  *cudaEvent\_t* start, stop;  CUDA\_CHECK(cudaEventCreate(&start));  CUDA\_CHECK(cudaEventCreate(&stop));   float alpha = *1.0*f, beta = *0.0*f;  std::vector<float> times; for(int i=*0*;i<*2*;i++){  *// Forward* CUBLAS\_CHECK(cublasGemmEx(handle,  CUBLAS\_OP\_N, CUBLAS\_OP\_N,  BATCH, HIDDEN\_DIM, IN\_DIM,  &alpha,  d\_X, CUDA\_R\_32F, BATCH,  d\_W1, CUDA\_R\_32F, IN\_DIM,  &beta,  d\_Z1, CUDA\_R\_32F, BATCH,  CUDA\_R\_32F, CUBLAS\_GEMM\_DEFAULT\_TENSOR\_OP));  int threads=*256*; int blocks=(BATCH\*HIDDEN\_DIM+threads-*1*)/threads;  add\_bias\_relu<<<blocks,threads>>>(d\_Z1, d\_b1, BATCH, HIDDEN\_DIM);  CUBLAS\_CHECK(cublasGemmEx(handle,  CUBLAS\_OP\_N, CUBLAS\_OP\_N,  BATCH, OUT\_DIM, HIDDEN\_DIM,  &alpha,  d\_Z1, CUDA\_R\_32F, BATCH,  d\_W2, CUDA\_R\_32F, HIDDEN\_DIM,  &beta,  d\_Ypred, CUDA\_R\_32F, BATCH,  CUDA\_R\_32F, CUBLAS\_GEMM\_DEFAULT\_TENSOR\_OP));  } for(int it=*0*; it<*10*; ++it) {  CUDA\_CHECK(cudaEventRecord(start));   *// Forward* CUBLAS\_CHECK(cublasGemmEx(handle,  CUBLAS\_OP\_N, CUBLAS\_OP\_N,  BATCH, HIDDEN\_DIM, IN\_DIM,  &alpha,  d\_X, CUDA\_R\_32F, BATCH,  d\_W1, CUDA\_R\_32F, IN\_DIM,  &beta,  d\_Z1, CUDA\_R\_32F, BATCH,  CUDA\_R\_32F, CUBLAS\_GEMM\_DEFAULT\_TENSOR\_OP));  int threads=*256*; int blocks=(BATCH\*HIDDEN\_DIM+threads-*1*)/threads;  add\_bias\_relu<<<blocks,threads>>>(d\_Z1, d\_b1, BATCH, HIDDEN\_DIM);  CUBLAS\_CHECK(cublasGemmEx(handle,  CUBLAS\_OP\_N, CUBLAS\_OP\_N,  BATCH, OUT\_DIM, HIDDEN\_DIM,  &alpha,  d\_Z1, CUDA\_R\_32F, BATCH,  d\_W2, CUDA\_R\_32F, HIDDEN\_DIM,  &beta,  d\_Ypred, CUDA\_R\_32F, BATCH,  CUDA\_R\_32F, CUBLAS\_GEMM\_DEFAULT\_TENSOR\_OP));    CUDA\_CHECK(cudaEventRecord(stop));  CUDA\_CHECK(cudaEventSynchronize(stop));  float ms;  CUDA\_CHECK(cudaEventElapsedTime(&ms, start, stop));  times.push\_back(ms);  }   std::cout << "CUDA + cuBLAS ms per iter:*\n*";  for(auto t: times) std::cout<<t<<" ";  std::cout<<"*\n*";   cudaFree(d\_X); cudaFree(d\_Z1); cudaFree(d\_Ypred);  cudaFree(d\_W1); cudaFree(d\_b1); cudaFree(d\_W2); cudaFree(d\_b2);  cublasDestroy(handle);  return *0*; } |

Листинг 4. nn\_cuda.cu