Implementacija okvira za izradu neuronskih mreža na Cuda platformi

Brekalo, Branimir

Sveučilište Josipa Jurja Strossmayera u Osijeku
Fakultet primijenjene matematike i informatike
Sveučilišni prijediplomski studij Matematika i računarstvo

Branimir Brekalo

Implementacija okvira za izradu neuronskih mreža na CUDA platformi

Završni rad

Osijek, 2023.

Sveučilište Josipa Jurja Strossmayera u Osijeku
Fakultet primijenjene matematike i informatike
Sveučilišni prijediplomski studij Matematika i računarstvo

Branimir Brekalo

Implementacija okvira za izradu neuronskih mreža na CUDA platformi

Završni rad

Mentor: izv. prof. dr. sc. Domagoj Matijević
Komentor: Antonio Jovanović

Osijek, 2023.

Sažetak: U ovom ćemo radu pokazat ćemo kako je moguće implementirati okvir za izradu neuronskih mreža na CUDA platformi i u okviru rada implementirati proizvoljnu mrežu i testirati ju. Najprije ćemo se upoznati sa osnovnom arhitekturom grafičkih kartica i CUDA platformom. Nakon toga pokazat ćemo sve potrebne korake za implementaciju neuronskih mreža, komentirajući njihovu primjenu i razlog implementacije.

Ključne riječi: CUDA, neuronske mreže, C++

Framework for creating neural networks on the CUDA platform

Abstract: In this paper, we will demonstrate how it is possible to implement a framework for creating neural networks on the CUDA platform, and within the scope of the paper, we will implement a custom network and test it. First, we will familiarize ourselves with the basic architecture of graphics cards and the CUDA platform. Following that, we will illustrate all the steps and necessary components for the implementation of neural networks, commenting on their application and the rationale behind their implementation.

Keywords: CUDA, neural networks, C++

Sadržaj

1. Uvod

Neuronske mreže, inspirirane biološkim neuronskim sustavima, matematički su modeli koji pomažu računalima da nauče i donose zaključke na temelju podataka. Ove mreže sastoje se od slojeva neurona, gdje svaki neuron prima ulazne podatke, obrađuje ih i prosljeđuje dalje. Ovisno o arhitekturi mreže, neuronske mreže mogu izvoditi različite zadatke poput klasifikacije, regresije, generiranja sadržaja, prevođenja jezika i još mnogo toga.

U današnjem digitalnom dobu, primjene neuronskih mreža su sveprisutne i revolucionarne. Od prepoznavanja uzoraka i obrade slika do prirodnog jezika i analize podataka, neuronske mreže postale su temelj tehnološkog napretka. Unatoč njihovoj velikoj primjenjivosti, složenost i zahtjevna obrada podataka često ostaju izazov za razumijevanje i implementaciju. Kada se upuštamo u učenje o neuronskim mrežama, često smo suočeni s nedostatkom dubokog razumijevanja kako se stvari zapravo izvršavaju.

Kao korisnici često se oslanjamo na već izrađene biblioteke i visoko apstraktne okvire koje jezici kao što je Python nude. Ove biblioteke omogućuju brzo kreiranje, treniranje i testiranje neuronskih mreža bez potrebe da se dublje zaranja u unutarnje mehanizme. Dok ovo olakšava razvoj i omogućuje širokom spektru korisnika da se uključi u izradu neuronskih mreža, često znači da ne ulazimo u stvarnu implementaciju algoritama ili detalje kako se operacije izvode na razini hardvera. Ovakav pristup, iako vrlo praktičan, može ograničiti naše razumijevanje neuronskih mreža.

U ovom radu prikazat će se izrada jednog okvira za izradu takvih mreža s pripadnim algoritmima i primjerima. Cijeli programski kod nalazi se na github repozitoriju.

2. Grafičke kartice i CUDA platforma

U današnjem tehnološkom okruženju, grafičke kartice više nisu samo sredstvo za prikazivanje grafičkih sadržaja i podršku 3D igrama. Njihova izvanredna sposobnost za obavljanje paralelnih izračuna igra ključnu ulogu u procvatu područja dubokog učenja i umjetne inteligencije. Pravilno iskorištavanje ovih potencijala znatno doprinosi napretku neuronskih mreža, čineći ih moćnijima i učinkovitijima. Prije nego što se upustimo u implementaciju neuronskih mreža na CUDA platformi, ključno je temeljito razumjeti prednosti grafičkih kartica i ulogu CUDA programskog okvira u ovom kontekstu.

2.1. Prednosti grafičkih kartica

Grafička kartica (GPU) pruža značajno veću propusnost uputa i propusnost memorije u usporedbi s procesorskom jedinicom (CPU) unutar sličnog cjenovnog i energetskog okvira. Mnoge aplikacije koriste ove mogućnosti kako bi brže radile na GPU-u nego na CPU-u. Ostali računalni uređaji, poput FPGA-ova, također su vrlo energetski učinkoviti, ali nude puno manju fleksibilnost programiranja u usporedbi s GPU-ima.

Ova razlika u sposobnostima između GPU-a i CPU-a postoji jer su dizajnirani s različitim ciljevima. Dok je CPU dizajniran da se izvodi niz operacija, nazvan nit (*thread*), što je brže moguće i može izvršiti nekoliko desetaka niti paralelno, GPU je dizajniran da se istovremeno izvodi tisuće takvih (amortizirajući sporiju izvedbu jedne niti kako bi se postigla veća propusnost).

GPU je specijaliziran za izuzetno paralelna računanja i stoga je dizajniran tako da se više tranzistora posvećuje obradi podataka umjesto predmemoriranju podataka i upravljanju protokom podataka. Aplikacije s visokim stupnjem paralelnosti mogu iskoristiti ovu izuzetno paralelnu prirodu GPU-a kako bi postigle veću izvedbu nego na CPU-u.

2.2. CUDA

CUDA dolazi s softverskim okruženjem koje omogućava programerima korištenje C++ kao *high-level* programskog jezika. NVIDIA ju je prvi put predstavila 2006. godine, a osim C++ programskog jezika podržava još nekoliko jezika kao što su C, Java i Python. Također sadrži biblioteke i *middleware* kao što su cuDNN, TensorRT i ostale.

2.2.1. Jezgre

CUDA proširuje C++ omogućujući programeru definiranje C++ funkcija, nazvanih **jezgrama** (engl. kernels), koje se, kada su pozvane, izvode N puta paralelno pomoću N različitih CUDA niti, za razliku od samo jednog izvođenja kao kod običnih C++ funkcija.

Jezgra se definira pomoću deklaracijskog specifikatora __global__, a broj CUDA niti koje izvršavaju tu jezgru za određeni poziv jezgre određuje se pomoću nove sintakse <<<...>>> za konfiguraciju izvođenja. Svaka nit koja se izvodi jezgri dobiva jedinstveni identifikator koji je dostupan unutar jezgre putem ugrađenih varijabli.

Primjer 2.1. Zbrajanje dva 1D vektora veličine N.

```
__global__ void addVectors(float* A, float* B, float* C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    addVectors<<<1, N>>>(A, B, C);
    ...
}
```

U ovom primjeru koristimo ugrađenu varijablu threadIdx.x za zbrajanje 2 vektora. Varijabla threadId.x predstavlja jedinstveni identifikator trenutne niti unutar blokova niti duž x-osi. U 1D rešetki blokova niti, vrijednost threadIdx.x se kreće od 0 do veličine bloka minus jedan. Ova vrijednost često se koristi za indeksiranje nizova ili za obavljanje izračuna koji bi trebali biti izvedeni odvojeno od strane svake niti u bloku. Varijabla blockIdx.x predstavlja jedinstveni identifikator trenutnog bloka isto kao i kod niti, dok blockDim.x je dimenzija bloka u smjeru x-osi.

Kernel pozivamo u main dijelu koda tako da prvi broj predstavlja broj blokova, dok drugi broj predstavlja broj niti po bloku. U našem slučaju pozvali smo 1 blok sa N niti u bloku.

Primjer 2.2. Zbrajanje dva 2D vektora veličine NxN.

```
__global__ void addVectors2D(float A[N][N], float B[N][N],
                             float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    addVectors2D<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

U ovom primjeru, definiraju se dimenzije bloka niti koji će se koristiti za izvođenje operacije. U ovom slučaju, svaki blok niti sastoji se od 16 niti po x-osi i 16 niti po y-osi, što ukupno čini 256 niti po bloku. Logika ostaje slična kao u 1D primjeru, ali se uvode dvije dimenzije. Zbrajanje dvaju matrica koristi 1D logiku zbrajanja jer se matricu može zamisliti kao dugačak vektor vrijednosti.

2.2.2. Jezgre u projektu

Tijekom cijelog projekta koristimo dogovorene veličine blokova i broja niti, zajedno s logikom za izračunavanje potrebnog broja blokova za izvršavanje. Svaki blok može podržati maksimalno 1024[1] niti, što ćemo u ovom slučaju iskoristiti. Međutim, kako bismo spriječili prevelik broj blokova, dinamički ćemo prilagoditi broj blokova prema veličini matrice s kojom radimo.

```
dim3 block_size(1024);
dim3 num_of_blocks((ukupan_broj_podataka + block_size.x - 1)
                   / block_size.x);
pozivJezgre<<<num_of_blocks, block_size>>>(argumenti);
```

Ovaj kod primjenjuje dinamičko skaliranje broja blokova kako bi se osiguralo da svi elementi budu obrađeni. Također se koristi maksimalna veličina bloka (1024 niti) koja je podržana na odabranoj grafičkoj kartici.

```
dim3 block_size(32, 32);
dim3 num_of_blocks((podaci.dims.x + block_size.x - 1) / block_size.x,
                   (podaci.dims.y + block_size.y - 1) / block_size.y);
pozivJezgre<<<num_of_blocks, block_size>>>(argumenti);
```

Ovaj kod primjenjuje isti princip kao i prethodni primjer, ali sada se koristi 2D mreža blokova kako bi se manipuliralo s dvodimenzionalnim podacima. Važno je osigurati da umnožak brojeva u varijabli block size ne prelazi 1024, što je ograničenje za maksimalni broj niti po bloku na većini grafičkih kartica. Ovaj kod se često koristi za operacije poput množenja dviju matrica.

[1] instaliranjem CUDA toolkit-a korisnik dobija demo datoteke za provjeru informacija o mogućnostima svoje grafičke kartice, pokretanjem deviceQuery.exe datoteke izlistavaju se sve bitne informacije tako i maksimalan broj niti po bloku, konkretno za ovaj projekt korištena je GeForce GTX 1050 Ti grafička kartica

3. Struktura projekta

Prije samog procesa implementacije, ključno je upoznati se s temeljnom strukturom projekta, relevantnim klasama i funkcijama koje će biti razvijene, te razumjeti svrhu njihove implementacije.

3.1. Dimenzije

Iako nije obavezan korak, razvili smo vlastitu strukturu za dimenzije koju ćemo koristiti za reprezentaciju dimenzija svih podataka u ovom projektu.

```
struct Dimensions
{
    size_t x, y;

    Dimensions(size_t x = 1, size_t y = 1);
};
```

Svaka instanca ove strukture ima dvije članice podataka: "x" i "y".

3.2. Matrica

U ovom projektu koristit ćemo matrice za reprezentaciju svih podataka. Sve matrice bit će implementirane kao pametni pokazivači (smart pointers). S obzirom na implementaciju pametnih pokazivača, omogućuje nam fleksibilno i učinkovito upravljanje memorijom za matrice u projektu.

```
class Matrix
{
private:
    bool isDeviceAllocated;
    bool isHostAllocated;

    void allocateCudaMemory();
    void allocateHostMemory();

public:
    Dimensions dims;

    std::shared_ptr<float> deviceData;
    std::shared_ptr<float> hostData;

    Matrix(size_t x_dim = 1, size_t y_dim = 1);
    Matrix(Dimensions dims);

    void allocateMemory();
```

```
    void allocateMemoryIfNotAllocated(Dimensions dims);

    void copyHostToDevice();
    void copyDeviceToHost();

    float& operator[](const int index);
    const float& operator[](const int index) const;

    void oneHotEncoding();

    bool deviceAllocation();
};
```

Klasa "Matrix" ima članove za alokaciju memorije na uređaju (GPU) i domaćinu (CPU), te član za dimenzije matrice. Također, koristi se pametni pokazivač "shared ptr<float>" za pohranu podataka na uređaju (deviceData) i domaćinu (hostData).

void allocateMemory() - alocira memorije na uređaju (GPU)

void allocateMemoryIfNotAllocated(Dimensions dims) - alocira memoriju na uređaju (GPU) sa određenim dimenzijama

copyHostToDevice() - kopira podatke sa CPU-a na GPU

copyDeviceToHost() - kopira podatke sa GPU-a na CPU

oneHotEncoding() - provodi "one-hot encoding" nad podacima po stupcima matrice, traži najveću vrijednost u stupcu i nju postavlja na 1, dok ostale postaju 0, s pretpostavkom da su vrijednosti između 0 i 1

deviceAllocation() - vraća TRUE ako su podaci na GPU-u, inače FALSE

3.3. Neuronska mreža

Klasa "NeuralNetwork" predstavlja centralnu komponentu koja integrira sve ključne dijelova neuronske mreže. Sprema informacije o slojevima mreže, funkciji troška, optimizacijskom algoritmu i regularizaciji. Ova klasa omogućuje izvođenje prolaza unaprijed (forward pass) i prolaz unatrag za optimizaciju težina (backpropagation).

Dodatno, klasa pruža funkcionalnost za izračunavanje preciznosti modela, što je često bitan kriterij za procjenu učinkovitosti neuronske mreže u problemima klasifikacije.

U osnovi, ova klasa predstavlja kompleksnu apstrakciju za upravljanje i treniranje neuronskih mreža, čineći proces razvoja i evaluacije modela učinkovitijim i organiziranim.

```
class NeuralNetwork
{
private:
    std::vector<NNLayer*> layers;

    Matrix Y, dY;

    float learning_rate;

    CostFunction* costFunction;

    Optimizer* optimizer;

    Regularization* regularization = nullptr;

public:
    NeuralNetwork(CostFunction* costFunction, Optimizer* optimizer,
    Regularization* regularization = nullptr, float learning_rate = 0.01);

    ~NeuralNetwork();

    void setCostFunction(CostFunction* costFunction);

    Matrix forward(Matrix X);
    void backprop(Matrix predictions, Matrix target);

    void addLayer(NNLayer *layer);
    std::vector<NNLayer*> getLayers() const;

    float computeAccuracy(Matrix& predictions, Matrix& targets);
};
```

void setCostFunction(CostFunction* costFunction) - postavlja funkciju troška za tu mrežu

Matrix forward(Matrix X) - provodi prolaz unaprijed (forward pass)

void backprop(Matrix predictions, Matrix target) - provodi prolaz unatrag i ažurira težine (backpropagation)

void addLayer(NNLayer *layer) - dodaje sloj u mrežu

vector<NNLayer*> getLayers() const - vraća vektor svih slojeva

float computeAccuracy(Matrix& predictions, Matrix& targets) - računa točnost s obzirom na predviđene vrijednosti i točne vrijednosti

3.4. Slojevi

Nadalje, predstavit ćemo osnovnu strukturu klasa za različite dijelove neuronske mreže, budući da svaka od tih klasa dijeli zajedničke funkcije.

Klasa "NNLayer" predstavlja jedan sloj u neuronskoj mreži, a u ovom kontekstu, čak i aktivacijske funkcije smatraju se slojevima. Svaki sloj ima svoje jedinstveno ime i dvije ključne funkcije, "forward" i "backprop", koje su od vitalnog značaja za provođenje prolaza kroz sloj i povratnu propagaciju u procesu učenja.

```
class NNLayer
{
protected:
    std::string name;

public:
    virtual ~NNLayer() = 0;

    virtual Matrix& forward(Matrix& A) = 0;
    virtual Matrix& backprop(Matrix& dZ, float learning_rate) = 0;

    std::string getName() { return this->name; };
};

inline NNLayer::~NNLayer() {}
```

3.4.1. Linearni sloj

Jedini sloji koji ima dodatne funkcije zbog svoje složenosti je klasa "LinearLayer" koja predstavlja linearni sloj [1, p. 17] u neuronskoj mreži ili jako povezan sloj. Linearni sloj je temeljni sloj koji provodi linearnu kombinaciju ulaza, što je matematički izraženo kao $Z = W \cdot A + b$, gdje je W matrica težina, A ulazni vektor, b vektor pomaka (bias), a Z izlazni vektor. Osim toga, u sebi sadrži pripadne gradijente težine, vektor pomaka i izlaza.

Također, omogućuje postavljanje optimizatora i regularizacije za prilagodbu učenja i reguliranje težina ako je potrebno.

Ovaj linearni sloj je ključan za neuronske mreže jer provodi osnovnu linearnu transformaciju ulaznih podataka, a zatim se obično kombinira s aktivacijskim slojem kako bi se modelirale nelinearne značajke u podacima.

```
class LinearLayer : public NNLayer
{
private:
    Matrix W, b, Z, A, dA, dW, db;

    Optimizer* optimizer;
```

```
    Regularization* regularization;

    void initializeBiasWithZeros();
    void initializeWeightsRandomly();

    void computeStoreBackpropError(Matrix& dZ);
    void computeStoreLayerOutput(Matrix& A);
    void computeStoreWGradient(Matrix& dZ);
    void computeStoreBGradient(Matrix& dZ);

public:
    LinearLayer(std::string name, Dimensions W_dims);
    ~LinearLayer();

    Matrix& forward(Matrix& A);
    Matrix& backprop(Matrix& dZ, float learning_rate = 0.01);

    int getXDim() const;
    int getYDim() const;

    Matrix getWeightsMatrix() const;
    Matrix getBiasVector() const;

    void setOptimizer(Optimizer* optimizer);
    void setRegularization(Regularization* regularization);

};
```

3.5. Funkcije troška

Funkcije troška (engl. cost functions) [1, p. 14] su matematičke funkcije koje se koriste u strojnom učenju i dubokom učenju kako bi se mjerila razlika između stvarnih vrijednosti (ground truth) i predviđenih vrijednosti modela. Osnovna svrha funkcija troška je kvantificirati koliko su predviđeni rezultati modela bliski stvarnim podacima.

Općenito, funkcije troška prihvaćaju stvarne vrijednosti (npr. oznake u problemima klasifikacije) i predviđene vrijednosti generirane modelom te izračunavaju numerički trošak ili gubitak koji proizlazi iz razlike između stvarnih i predviđenih vrijednosti. Cilj je minimizirati ovaj trošak tijekom treninga kako bi se model naučio bolje se prilagoditi podacima.

```
class CostFunction
{
public:
    virtual float cost(Matrix& target, Matrix& predicted,
                       Matrix& W) = 0;
    virtual Matrix dCost(Matrix& predicted, Matrix& target,
                         Matrix& dY) = 0;
};
```

Za svaku funkciju troška, bitno je imati samo informaciju o trošku u odnosu na predviđene i stvarne vrijednosti, kao i njezinu derivaciju kako bismo mogli ispravno ažurirati težine.

3.6. Optimizacijski algoritmi

Optimizacijski algoritmi [1, p. 134-141] su ključni elementi u treniranju neuronskih mreža i drugih modela u strojnom učenju. Njihova svrha je prilagoditi težine modela kako bi se minimizirala funkcija troška, čime se postiže bolja prilagodba podacima i poboljšava performansa modela.

Svaka klasa optimizatora ima određene dimenzije parametara koje optimizira, te implementira metode za ažuriranje težina i pomaka koristeći gradijente i brzinu učenja.

```
class Optimizer
{
public:
    Dimensions wDims, bDims;

    virtual void updateW(Matrix &dW, Matrix &W,
                         float learning_rate) = 0;
    virtual void updateB(Matrix &db, Matrix &b,
                         float learning_rate) = 0;
    virtual void updateStep(Matrix &dW, Matrix &W, Matrix &db,
                            Matrix &b, float learning_rate) = 0;
    virtual void initialize(Dimensions wDims, Dimensions bDims) { };
};
```

virtual void initialize(Dimensions wDims, Dimensions bDims) - ako optimizacijski alogritam sprema neke vrijednosti kroz više prolaza (npr. Adam), za svaki sloj se inicijaliziraju matrice i zauzima potrebna memorija

3.7. Regularizacija

Regularizacija [1, p. 181] je tehnika koja se koristi tijekom treninga neuronskih mreža i drugih modela strojnog učenja kako bi se spriječilo prenaučenost (engl. overfitting) i poboljšala generalizacija modela na nepoznate podatke. Osnovna svrha regularizacije je kontrolirati kompleksnost modela tako da se izbjegne prekomjerno prilagođavanje trening podacima.

Regularizacija je važna jer pomaže u održavanju ravnoteže između sposobnosti modela da nauči kompleksne obrasce u podacima i sposobnosti modela da generalizira na novim, nepoznatim podacima.

Svaka regularizacija sadrži regularizaciju na gradijent težina (dW) tijekom treninga i doprinos regularizacije ukupnom trošku (cost) modela.

```
class Regularization
{
public:
    virtual void gradientRegularization(Matrix& W,
                                        Matrix &dW, int size) = 0;
    virtual float costRegularization(Matrix &W) = 0;
};
```

4. Implementacija

U implementacijskom segmentu fokusirat ćemo se na stvaranje CUDA jezgri i objasniti logiku koja stoji iza funkcija i klasa koje smo ranije spomenuli.

4.1. Dimenzije

```
Dimensions::Dimensions(size_t x, size_t y) : x(x), y(y) { }
```

4.2. Matrica

```
Matrix::Matrix(size_t x_dim, size_t y_dim) :
    dims(x_dim, y_dim),
    deviceData(nullptr),
    hostData(nullptr),
    isDeviceAllocated(false),
    isHostAllocated(false)
{ }

Matrix::Matrix(Dimensions dims) :
    Matrix(dims.x, dims.y)
{ }

void Matrix::allocateCudaMemory()
{
    if (!isDeviceAllocated)
    {
        float* device_memory = nullptr;
        cudaMalloc(&device_memory, dims.x * dims.y * sizeof(float));
        deviceData = std::shared_ptr<float>(device_memory, [&](float* ptr)
        { cudaFree(ptr); });
        isDeviceAllocated = true;
    }
}

void Matrix::allocateHostMemory()
{
    if (!isHostAllocated)
    {
        hostData = std::shared_ptr<float>(new float[dims.x * dims.y],
        [&](float* ptr) { delete[] ptr; });
        isHostAllocated = true;
    }
}

void Matrix::allocateMemory()
```

```
{
    allocateCudaMemory();
    allocateHostMemory();
}

void Matrix::allocateMemoryIfNotAllocated(Dimensions dims)
{
    if (!isDeviceAllocated && !isHostAllocated)
    {
        this->dims = dims;
        allocateMemory();
    }
}

void Matrix::copyHostToDevice()
{
    if (isDeviceAllocated && isHostAllocated)
    {
        cudaMemcpy(deviceData.get(), hostData.get(), dims.x * dims.y
        * sizeof(float),  cudaMemcpyHostToDevice);
    }
}

void Matrix::copyDeviceToHost()
{
    if (isDeviceAllocated && isHostAllocated)
    {
        cudaMemcpy(hostData.get(), deviceData.get(), dims.x * dims.y
        * sizeof(float), cudaMemcpyDeviceToHost);
    }
}

float& Matrix::operator[](const int index)
{
    return hostData.get()[index];
}

const float& Matrix::operator[](const int index) const
{
    return hostData.get()[index];
}

void Matrix::oneHotEncoding()
{
    for(int col = 0; col < dims.x; col++)
    {
        float max = -1.0f;
        int maxInd = -1;
```

```
            for(int row = 0; row < dims.y; row++)
            {
                float current = hostData.get()[col + dims.x * row];
                if(current > max)
                {
                    max = current;
                    maxInd = col + dims.x * row;
                }
            }

            for(int row = 0; row < dims.y; row++)
            {
                if(col + dims.x * row == maxInd)
                    hostData.get()[col + dims.x * row] = 1.0f;
                else
                    hostData.get()[col + dims.x * row] = 0.0f;
            }
        }
}

bool Matrix::deviceAllocation() { return this->isDeviceAllocated; }
```

U kodu matrice, važno je napomenuti kako smo implementirali upravljanje memorijom. U oba slučaja koristimo pametne pokazivače koji automatski oslobađaju memoriju kada više nije potrebna. U λ izrazima definiramo funkciju koja će se izvršiti kada pametni pokazivač više ne bude u upotrebi.

4.3. Neuronska mreža

```
NeuralNetwork::NeuralNetwork(CostFunction* costFunction,
    Optimizer* optimizer, Regularization* regularization ,
    float learning_rate) :
        costFunction(costFunction),
        optimizer(optimizer),
        regularization(regularization),
        learning_rate(learning_rate)
{ }

NeuralNetwork::~NeuralNetwork()
{
    for (auto layer : layers)
        delete layer;
}

void NeuralNetwork::setCostFunction(CostFunction* costFunction)
{
```

```
    this->costFunction = costFunction;
}

void NeuralNetwork::addLayer(NNLayer* layer)
{
    this->layers.push_back(layer);

    LinearLayer* linearLayer = dynamic_cast<LinearLayer*>(layer);


    if (linearLayer)
    {
        Optimizer* newOptimizer = nullptr;
        AdamOptimizer* adamOptimizer =
        dynamic_cast<AdamOptimizer*>(optimizer);

        if(adamOptimizer)
        {
            newOptimizer = new AdamOptimizer(adamOptimizer->getBeta1(),
            adamOptimizer->getBeta2(), adamOptimizer->getEpsilon());
            newOptimizer->initialize(Dimensions(linearLayer->getXDim(),
            linearLayer->getYDim()), Dimensions(linearLayer->getXDim(), 1));
        }
        else
        {
            newOptimizer = optimizer;
        }

        linearLayer->setOptimizer(newOptimizer);
        linearLayer->setRegularization(regularization);
        }
}

Matrix NeuralNetwork::forward(Matrix X)
{
    Matrix Z = X;

    for (auto layer : layers)
        Z = layer->forward(Z);

    Y = Z;

    return Y;
}

void NeuralNetwork::backprop(Matrix predictions, Matrix target)
{
    dY.allocateMemoryIfNotAllocated(predictions.dims);
```

```
    Matrix error = costFunction->dCost(predictions, target, dY);

    for (auto it = this->layers.rbegin(); it != this->layers.rend(); it++)
        error = (*it)->backprop(error, learning_rate);

    cudaDeviceSynchronize();
}

std::vector<NNLayer*> NeuralNetwork::getLayers() const { return layers; }

float NeuralNetwork::computeAccuracy(Matrix& predictions, Matrix& target)
{
    assert(predictions.dims.x == target.dims.x
    && predictions.dims.y == target.dims.y);

    if(predictions.deviceAllocation())
        predictions.copyDeviceToHost();

    predictions.oneHotEncoding();

    int correct = 0;

    for(int col = 0; col < target.dims.x; col++)
    {
        int flag = 1;
        for(int row = 0; row < target.dims.y; row++)
        {
            if(predictions[col + target.dims.x * row] !=
               target[col + target.dims.x * row])
                flag = 0;
           }

        if(flag)
            correct++;
    }

    return static_cast<float>(correct) / target.dims.x;
}
```

U ovoj implementaciji, posebno je važno naglasiti logiku koja stoji iza funkcije backprop. Backpropagation prvo izračunava grešku na izlazu neuronske mreže, a zatim tu grešku unazad prosljeđuje svim slojevima mreže. Svaki sloj također ima svoju backpropagation funkciju koja provodi određene operacije. Ova implementacija odgovara pravilu derivacije složene funkcije (engl. chain rule), što je ključni princip diferencijalnog računa, primijenjenog na neuronske mreže. Ovaj pristup omogućuje automatsko računanje gradijenata i ažuriranje parametara mreže, što značajno pojednostavljuje proces treniranja.

Također, funkcija koja dodaje sloj trenutno se brine o Adam optimizacijskom algoritmu. Ako je Adam algoritam postavljen za ovu mrežu, svaki linearni sloj dobiva svoju vlastitu instancu Adama. To je zato što se u svakoj instanci Adama čuvaju dodatne matrice koje se koriste prilikom ažuriranja težina u svakom sloju.

4.4. Slojevi

4.4.1. Linearni sloj

Linearni sloj, također poznat kao jako povezani sloj, koristan je u neuronskim mrežama jer obavlja transformaciju ulaznih podataka pomoću težina (W) i vektora pomaka (b).

Linearni sloj je osnovna komponenta u dubokim neuronskim mrežama i omogućava modelima da nauče složene značajke iz ulaznih podataka tako da prilagode težine i vektore pomaka tijekom treniranja. Ovaj sloj često zahtijeva optimizacijske algoritme za ažuriranje težina i vektora pomaka kako bi se minimizirala funkcija troška i poboljšala točnost modela. Također, mogu se primijeniti različite tehnike regularizacije kako bi se spriječilo prenaučivanje i poboljšala generalizacija modela.

Propagacija unaprijed i izračun izlaza Z:

$$Z = W \cdot A + b$$

Povratna propagacija i pripadni gradijenti (gradijent izlaza dZ, gradijent ulaza dA, gradijent težina dW, gradijent vektora pomaka db):

Izračun gradijenta dA:

$$dA = W^T \cdot dZ$$

Izračun gradijenta težina dW:

$$dW = \frac{1}{m} \cdot dZ \cdot A^T$$

Izračun gradijenta vektora pomaka db:

$$db = \frac{1}{m} \cdot \sum_{i=1}^{m} dZ^{(i)}$$

4.4.2. ReLU sloj/aktivacija

ReLU (Rectified Linear Unit) [1, p. 13] aktivacijska funkcija je jedna od najčešće korištenih aktivacijskih funkcija u neuronskim mrežama. Njena formula i derivacija su sljedeće:

$$\text{ReLU}(x) = \max(0, x)$$

$$\frac{d\text{ReLU}(x)}{dx} = \begin{cases} 1, & \text{ako } x > 0 \\ 0, & \text{ako } x \leq 0 \end{cases}$$

Prednosti ReLU aktivacijske funkcije uključuju brzu konvergenciju tijekom treniranja neuronskih mreža zbog nepostojanja zasićenja gradijenta za pozitivne vrijednosti i jednostavnu implementaciju.

Međutim, mana ReLU-a je da može uzrokovati nestajući gradijent za negativne vrijednosti, što može otežati treniranje dubokih mreža, a također ima problem "mrtvih neurona" gdje se neuroni mogu blokirati i ne aktivirati tijekom treniranja.

U praksi, ReLU se često koristi kao aktivacijska funkcija za skriveni slojevi u dubokim neuronskim mrežama zbog svoje jednostavnosti i brzine treniranja. Međutim, postoje varijacije ReLU aktivacijske funkcije, poput Leaky ReLU i Parametric ReLU, koje pokušavaju riješiti problem nestajućeg gradijenta kod negativnih ulaza.

```
__global__ void reluActivationForward(float* Z, float *A, int Z_x_dim,
                                      int Z_y_dim)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;

        if (index < Z_x_dim * Z_y_dim)
    {
            A[index] = fmaxf(Z[index], 0);
        }
}

__global__ void reluActivationBackprop(float* Z, float *dA, float* dZ,
                                       int Z_x_dim, int Z_y_dim)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;

        if (index < Z_x_dim * Z_y_dim)
    {
            if (Z[index] > 0)
                dZ[index] = dA[index];
            else
                dZ[index] = 0;
        }
}

ReLUActivation::ReLUActivation(std::string name) { this->name = name; }

ReLUActivation::~ReLUActivation() { }
```

```
Matrix& ReLUActivation::forward(Matrix& Z)
{
    this->Z = Z;

    A.allocateMemoryIfNotAllocated(Z.dims);

    dim3 block_size(1024);
    dim3 num_of_blocks(
    (Z.dims.y * Z.dims.x + block_size.x - 1) / block_size.x);

    reluActivationForward<<<num_of_blocks, block_size
    (Z.deviceData.get(), A.deviceData.get(), Z.dims.x, Z.dims.y);

    return A;
}

Matrix& ReLUActivation::backprop(Matrix& dA, float learning_rate)
{
    dZ.allocateMemoryIfNotAllocated(Z.dims);

    dim3 block_size(1024);
    dim3 num_of_blocks((Z.dims.y * Z.dims.x + block_size.x - 1) / block_size.x

    reluActivationBackprop<<<num_of_blocks, block_size>>>
    (Z.deviceData.get(), dA.deviceData.get(),
    dZ.deviceData.get(), Z.dims.x, Z.dims.y);

    return dZ;
}
```

4.4.3. Softmax aktivacija

Softmax [1, p. 14] je aktivacijska funkcija koja se često koristi u neuronskim mrežama, posebno za višeklasnu klasifikaciju. Njegova svrha je pretvoriti ulazne vrijednosti (koje se često zovu "logits") u vjerojatnosti tako da svaka izlazna vrijednost predstavlja vjerojatnost da pripada određenoj klasi.

Softmax funkcija definirana je sljedećom formulom:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

gdje je z_i ulazna vrijednost za i-tu klasu, K je ukupan broj klasa.

Numerically stable softmax je tehnika koja se koristi za poboljšanje numeričke stabilnosti prilikom izračuna softmax vjerojatnosti. Problematično je kada su ulazne vrijednosti z_i velike, što može uzrokovati overflow u numeričkom računanju.

Umjesto izračunavanja softmaxa prema originalnoj formuli, koristi se modificirana verzija koja izbjegava overflow tako da izračuna softmax na način da oduzme maksimalnu ulaznu vrijednost ($\max_i z_i$) od svih ulaznih vrijednosti prije računanja eksponencijalne funkcije:

$$\text{softmax}(z_i) = \frac{e^{z_i - \max_j z_j}}{\sum_{j=1}^{K} e^{z_j - \max_k z_k}}$$

Oduzimanje maksimalne vrijednosti ne mijenja relativne vjerojatnosti između klasa, ali čini izračun stabilnijim, jer eksponencijalne funkcije rastu eksponencijalno brže i može se smanjiti rizik od overflow-a.

Softmax log trick se koristio u implementaciji jer vrijednosti prije primjene softmax-a su velike na izlazu.

```
__global__ void softmaxActivationForward(float *Z, float *A,
                                         int Z_x_dim, int Z_y_dim)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (col < Z_x_dim)
    {
        float max_val = -FLT_MAX;
        for (int i = 0; i < Z_y_dim; i++)
            max_val = fmax(max_val, Z[i * Z_x_dim + col]);

        float exp_sum = 0.0f;
        for (int i = 0; i < Z_y_dim; i++)
        {
            A[i * Z_x_dim + col] = exp(Z[i * Z_x_dim + col] - max_val);
            exp_sum += A[i * Z_x_dim + col];
        }

        for (int i = 0; i < Z_y_dim; i++)
            A[i * Z_x_dim + col] /= exp_sum;
    }
}


SoftmaxActivation::SoftmaxActivation(std::string name)
                                    { this->name = name; }

SoftmaxActivation::~SoftmaxActivation() { }

Matrix &SoftmaxActivation::forward(Matrix &Z)
{
    this->Z = Z;

    A.allocateMemoryIfNotAllocated(Z.dims);
```

```
    dim3 block_size(32, 32);
    dim3 num_of_blocks(
    (Z.dims.x + block_size.x - 1) / block_size.x,
    (Z.dims.y + block_size.y - 1) / block_size.y);

    softmaxActivationForward<<<num_of_blocks, block_size>>>
    (Z.deviceData.get(), A.deviceData.get(), Z.dims.x, Z.dims.y);

    return A;
}

Matrix& SoftmaxActivation::backprop(Matrix& dA, float learning_rate)
{
    return dA;
}
```

4.5. Funkcije troška

4.5.1. Cross entropy funkcija troška

Višeklasna unakrsna entropija (Cross entropy) [1, p. 15], također poznata kao kategorička unakrsna entropija, često se koristi kao funkcija gubitka u strojnom učenju za zadatke klasifikacije u kojima je cilj dodijeliti podatke jednoj od više mogućih klasa. Ona mjeri različitost između predviđenih vjerojatnosti klasa i stvarnih oznaka klasa. U kontekstu dubokog učenja i neuronskih mreža, koristi se za kvantificiranje pogreške između predviđene vjerojatnosti raspodjele klasa i stvarnih oznaka kodiranje u one-hot obliku.
Formula za gubitak višeklasne unakrsne entropije, s obzirom na predviđanja (obično dobivena iz aktivacije softmax) i ciljne oznake (kodirane u obliku one-hot):

$$Loss = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{C} y_{ij}\log(p_{ij}+\epsilon)$$

i njena derivacija (ako je izlazni sloj u one-hot obliku, odnosno primjenjen je softmax):

$$dLoss = \frac{y_{ij}-p_{ij}}{m}$$

gdje je:

- m broj podataka,
- C broj klasa,
- y_{ij} one-hot enkodirane stvarne oznake za podatak i i klasu j,
- p_{ij} predviđena vjerojatnost da podatak i pripada klasi j,
- ϵ mala konstanta koja se dodaje kako bi se spriječila numerička nestabilnost prilikom uzimanja logaritma.

```
__global__ void crossEntropyCost(float* predictions, float* target,
                                 int features, int data, float* cost)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (col < data)
    {
        float col_cost = 0.0f;
        for (int row = 0; row < features; row++)
        {
            float epsilon = 1e-7f;
            col_cost += (-target[row * data + col] *
            log(predictions[row * data + col] + epsilon));
        }

        atomicAdd(cost, col_cost);
    }

    if (col == data - 1)
        atomicExch(cost, *cost / data);
}

__global__ void dCrossEntropyCost(float* predictions, float* target,
                                  float* dY, int features, int data)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (col < data)
    {
        for (int row = 0; row < features; row++)
        {
            int index = row * data + col;
            dY[index] = (predictions[index] - target[index]) / data;
        }
    }
}

float CrossEntropyCost::cost(Matrix& predictions, Matrix& target, Matrix& W)
{
    assert(predictions.dims.x == target.dims.x &&
    predictions.dims.y == target.dims.y);

    float* cost;

    cudaMallocManaged(&cost, sizeof(float));

    *cost = 0.0f;
```

```
    dim3 block_size(1024);
    dim3 num_of_blocks(
    (predictions.dims.x + block_size.x - 1) / block_size.x);

    crossEntropyCost<<<num_of_blocks, block_size>>>
    (predictions.deviceData.get(), target.deviceData.get(),
    predictions.dims.y, predictions.dims.x, cost);

    cudaDeviceSynchronize();

    float cost_value = *cost;

    cudaFree(cost);

    if (regularization != nullptr)
    {
        float regTerm = 0.0f;

        regTerm = regularization->costRegularization(W);

        cost_value += regTerm;
    }

    return cost_value;
}

Matrix CrossEntropyCost::dCost(Matrix& predictions, Matrix& target,
Matrix& dY)
{
    assert(predictions.dims.x == target.dims.x &&
    predictions.dims.y == target.dims.y);

    dim3 block_size(1024);
    dim3 num_of_blocks(
    (predictions.dims.x + block_size.x - 1) / block_size.x);

    dCrossEntropyCost<<<num_of_blocks, block_size>>>
    (predictions.deviceData.get(), target.deviceData.get(),
    dY.deviceData.get(), predictions.dims.y, predictions.dims.x);

    cudaDeviceSynchronize();

    return dY;
}

CrossEntropyCost::CrossEntropyCost(Regularization* regularization):
regularization(regularization) { }
```

4.6. Optimizacijski algoritmi

4.6.1. Adam

Adam (Adaptive Moment Estimation) [1, p. 140] je popularni optimizacijski algoritam koji se koristi za treniranje neuronskih mreža i druge optimizacijske probleme u strojnom učenju. Ovaj algoritam kombinira prednosti dvaju drugih poznatih optimizacijskih algoritama: AdaGrad i RMSprop.

Osnovna ideja iza Adama je prilagodljiva stopa učenja (learning rate) za svaki parametar modela. Algoritam održava dva pokretna prosjeka (momentume) za svaki parametar: prvi moment (srednja vrijednost gradijenta) i drugi moment (srednja vrijednost kvadrata gradijenta). Ova dva momenta se koriste za prilagodbu stope učenja kako bi se brže konvergiralo prema minimumu funkcije troška, a istovremeno izbjegle oscilacije i divergencije.

Ažuriranje težina:

$$W_{t+1} = W_t - \frac{\alpha}{\sqrt{v_t} + \epsilon} \cdot m_t$$

gdje su:

- W_t težine u trenutnom koraku t,
- α stopa učenja (learning rate),
- m_t prvi moment gradijenta,
- v_t drugi moment gradijenta,
- ϵ mala vrijednost za stabilnost.

Ažuriranje vektora pomaka:

$$b_{t+1} = b_t - \frac{\alpha}{\sqrt{v_t} + \epsilon} \cdot m_t$$

Za vektor pomaka vrijedi ista notacija, umjesto W pišemo b.

Adam je popularan izbor za optimizaciju neuronskih mreža zbog svoje sposobnosti brzog konvergiranja i prilagodbe stopa učenja za svaki parametar.

```
__global__ void updateWAdam(float *dW, float *W, float *mW, float *vW,
float beta1, float beta2, float epsilon, float learning_rate, int size,
int t)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < size)
    {
        mW[idx] = beta1 * mW[idx] + (1 - beta1) * dW[idx];
        vW[idx] = beta2 * vW[idx] + (1 - beta2) * dW[idx] * dW[idx];

        float beta1_t = pow(beta1, t);
        float beta2_t = pow(beta2, t);
```

```
        float m_hat = mW[idx] / (1 - beta1_t);
        float v_hat = vW[idx] / (1 - beta2_t);

        W[idx] -= learning_rate * m_hat / (sqrt(v_hat) + epsilon);
    }
}

__global__ void updateBAdam(float *db, float *b, float *mb, float *vb,
float beta1, float beta2, float epsilon, float learning_rate, int size,
int t)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < size)
    {

        mb[idx] = beta1 * mb[idx] + (1 - beta1) * db[idx];
        vb[idx] = beta2 * vb[idx] + (1 - beta2) * db[idx] * db[idx];


        float beta1_t = pow(beta1, t);
        float beta2_t = pow(beta2, t);


        float m_hat = mb[idx] / (1 - beta1_t);
        float v_hat = vb[idx] / (1 - beta2_t);
        b[idx] -= learning_rate * m_hat / (sqrt(v_hat) + epsilon);
    }
}

AdamOptimizer::AdamOptimizer(float beta1, float beta2, float epsilon):
    beta1(beta1),
    beta2(beta2),
    epsilon(epsilon),
    t(1)
{ }

void AdamOptimizer::setMatricesToZero()
{
    for (int x = 0; x < mW.dims.x; x++)
        for (int y = 0; y < mW.dims.y; y++)
            mW[y * mW.dims.x + x] = vW[y * mW.dims.x + x] = 0.0f;

    for (int x = 0; x < mb.dims.x; x++)
        for (int y = 0; y < mb.dims.y; y++)
            mb[y * mb.dims.x + x] = vb[y * mb.dims.x + x] = 0.0f;
```

```
    mW.copyHostToDevice();
    mb.copyHostToDevice();
}

void AdamOptimizer::initialize(Dimensions weightDims, Dimensions biasDims)
{
    mW = Matrix(weightDims);
    vW = Matrix(weightDims);
    mb = Matrix(biasDims);
    vb = Matrix(biasDims);

    mW.allocateMemory();
    vW.allocateMemory();
    mb.allocateMemory();
    vb.allocateMemory();

    setMatricesToZero();
}

void AdamOptimizer::updateW(Matrix &dW, Matrix &W, float learning_rate)
{
    dim3 block_size(1024);
    dim3 num_of_blocks(
    (dW.dims.y * dW.dims.x + block_size.x - 1) / block_size.x);

    updateWAdam<<<num_of_blocks, block_size>>>(dW.deviceData.get(),
    W.deviceData.get(), mW.deviceData.get(), vW.deviceData.get(),
    beta1, beta2, epsilon, learning_rate, dW.dims.y * dW.dims.x, t);

}

void AdamOptimizer::updateB(Matrix &db, Matrix &b, float learning_rate)
{
    dim3 block_size(1024);
    dim3 num_of_blocks(
    (db.dims.y * db.dims.x + block_size.x - 1) / block_size.x);

    updateBAdam<<<num_of_blocks, block_size>>>(db.deviceData.get(),
    b.deviceData.get(), mb.deviceData.get(), vb.deviceData.get(),
    beta1, beta2, epsilon, learning_rate, db.dims.y * db.dims.x, t);
}

void AdamOptimizer::updateStep(Matrix &dW, Matrix &W,
          Matrix &db, Matrix &b, float learning_rate)
{
    updateW(dW, W, learning_rate);
    updateB(db, b, learning_rate);
    increaseT();
```

```
}

void AdamOptimizer::increaseT() { t++; }

float AdamOptimizer::getBeta1() { return beta1; }

float AdamOptimizer::getBeta2() { return beta2; }

float AdamOptimizer::getEpsilon() { return epsilon; }
```

4.7. Regularizacija

4.7.1. L2 regularizacija

L2 regularizacija [1, p. 182], također poznata kao "Ridge regularizacija", je tehnika korištena u strojnom učenju kako bi se spriječila prenaučenost modela i poboljšala generalizacija na nepoznate podatke.

Osnovna ideja L2 regularizacije je dodavanje dodatnog člana u funkciju troška modela koji kažnjava velike težine. To se postiže dodavanjem kvadrata svih težina u funkciju troška, pomnoženih s faktorom regularizacije (λ ili α). Formula za L2 regularizaciju uključuje kvadrirane vrijednosti svih težina u modelu:

$$Loss = Loss + \frac{\lambda}{2 \cdot m} \cdot \|W\|_F$$

Gdje je $\| \cdot \|_F$ Frobenius-ova norma. [5, p. 50]

Kada se koristi L2 regularizacija, model će težiti prema manjim vrijednostima težina jer velike težine znače veći doprinos funkciji troška. Ovo pomaže u sprečavanju prenaučenosti jer model će biti skloniji naučiti općenite obrasce umjesto da se prekomjerno prilagodi trening podacima.

Vrijednost faktora regularizacije (λ ili α) kontrolira koliko će jak utjecaj imati L2 regularizacija na model. Veći faktor regularizacije će rezultirati većim kaznama za velike težine i stoga će model imati tendenciju prema manjim težinama. Važno je napomenuti da vrijednost faktora regularizacije treba pažljivo odabrati tijekom treninga kako bi se postigla ravnoteža između smanjenja prenaučenosti i zadržavanja dobre generalizacije modela.

```
__global__ void applyRegularization(float *dW, float *W,
                                    float lambda, int size)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < size)
    {
        dW[idx] += (lambda / size) * W[idx];
    }
}
```

```
__global__ void calculateRegularizationTerm(float* W, float lambda,
                                int size, float* regularizationTerm)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    float reg_cost = 0.0f;

    if (idx < size)
    {
        float w_i = W[idx];
        reg_cost += w_i * w_i;
    }

    atomicAdd(regularizationTerm, reg_cost);

}

void L2::gradientRegularization(Matrix& W, Matrix &dW, int size)
{
    dim3 block_size(1024);
    dim3 num_of_blocks((W.dims.x + block_size.x - 1) / block_size.x);

    applyRegularization<<<num_of_blocks, block_size>>>
    (dW.deviceData.get(), W.deviceData.get(),
    lambda, dW.dims.x * dW.dims.y);
}

float L2::costRegularization(Matrix &W)
{
    float* regularizationTerm;

    cudaMallocManaged(&regularizationTerm, sizeof(float));

    *regularizationTerm = 0.0f;

    dim3 block_size(1024);
    dim3 num_of_blocks((W.dims.x + block_size.x - 1) / block_size.x);

    calculateRegularizationTerm<<<num_of_blocks, block_size>>>
    (W.deviceData.get(), lambda,
    W.dims.x * W.dims.y, regularizationTerm);

    cudaDeviceSynchronize()

    float regularizationTermValue = *regularizationTerm;

    cudaFree(regularizationTerm);
```

```
    regularizationTermValue *= (lambda / (2 * W.dims.x));

    return regularizationTermValue;
}

L2::L2(float lambda): lambda(lambda) { }
```

5. Izgradnja, testiranje i usporedba mreže s drugim okvirima

Nakon što završimo implementaciju, sljedeći korak je stvaranje identičnih neuronskih mreža pomoću različitih okvira, te usporedba njihove uspješnosti na testnim podacima. Cilj je kreirati dvije potpuno identične mreže s istim parametrima i zatim provesti usporedbu njihove preciznosti s testnim podacima.

5.1. MNIST

MNIST je često korišten skup podataka koji se koristi kao jedan od prvih primjera u strojnom učenju. Sastoji se od 70 000 slika rukom pisanih brojeva od 0 do 9, svaka dimenzija 28x28 piksela, s odgovarajućim oznakama. Svaki piksel je predstavljen brojevima od 0 do 255, dok su oznake brojevi od 0 do 9. Zahvaljujući svojoj popularnosti, MNIST dataset postao je standardno mjerilo za usporedbu performansi različitih modela i tehnika strojnog učenja.

5.2. Izgradnja proizvoljne mreže - odabir parametara

Kreiranje neuronske mreže uključuje odabir različitih parametara koji će imati utjecaj na performanse mreže.

5.3. Funkcija troška, stopa učenja, regularizacija i optimizacijski algoritam

Za početak, trebamo odabrati prikladnu funkciju troška za ovaj klasifikacijski problem, stoga smo izabrali CrossEntropy loss funkciju. Također, u ovom primjeru nećemo primjenjivati regularizaciju, pa je vrijednost regularizacijskog parametra λ postavljena na 0. Stopu učenja postavljamo na 0.01, što je često korištena defaultna vrijednost. Nadalje, koristimo Adam optimizacijski algoritam s defaultnim postavkama. Na kraju, inicijaliziramo neuronsku mrežu s ovim parametrima.

```
float lambda = 0.00f;

float beta1 = 0.9f;
float beta2 = 0.999f;
float epsilon = 1e-8f;

float learning_rate = 0.01f;

L2 L2(lambda);

CrossEntropyCost crossEntropy(&L2);

AdamOptimizer adam(beta1, beta2, epsilon);

NeuralNetwork nn(&crossEntropy, &adam, &L2, learning_rate);
```

5.4. Slojevi i aktivacijske funkcije

Za ulazni sloj naše neuronske mreže koristimo 784 značajke, gdje svaka značajka predstavlja 1 piksel na slici. Svaki piksel je normaliziran kako bi vrijednosti bile unutar raspona od 0 do 1. Ulazni sloj obrađuje te podatke i transformira ih u prvi skriveni sloj s 128 neurona. Nad neuronima u ovom sloju primjenjuje se ReLU aktivacijska funkcija. Nakon toga, stvaramo još jedan skriveni sloj s 64 neurona, također s primjenom ReLU aktivacijske funkcije. Izlazni sloj naše mreže sastoji se od 10 neurona jer rješavamo klasifikacijski problem s 10 klasa. Kako bismo bolje interpretirali rezultate na izlaznom sloju, primjenjujemo Softmax aktivacijsku funkciju koja transformira izlaz u postotke koji predstavljaju vjerojatnost da slika pripada određenoj klasi.

```
nn.addLayer(new LinearLayer("linear1", Dimensions(784, 128)));
nn.addLayer(new ReLUActivation("relu"));
nn.addLayer(new LinearLayer("linear2", Dimensions(128, 64)));
nn.addLayer(new ReLUActivation("relu2"));
nn.addLayer(new LinearLayer("linear3", Dimensions(64, 10)));
nn.addLayer(new SoftmaxActivation("softmax"));
```

5.5. Treniranje mreže

Učitajmo skup podataka MNIST za obuku koji se sastoji od 60 000 primjera. Razdijelit ćemo podatke u 1875 grupa, pri čemu svaka grupa ima 32 primjera. Naša neuronska mreža će ukupno vidjeti sve podatke 10 puta (engl. epoch). Za svaku grupu, izračunat ćemo prolaz unaprijed (engl. forward pass), a zatim provesti algoritam povratne propagacije (engl. backpropagation) kako bismo ažurirali težine u mreži. Također, za svaku grupu izračunat ćemo vrijednost funkcije troška. Na kraju svake epohe, izračunat ćemo prosječni trošak temeljen na svim grupama podataka.

```
MNIST traindata(32, 1875, "datasets/mnist_train.csv");

Matrix Y;

for (int epoch = 0; epoch < 11; epoch++)
{
    float cost = 0.0;

    for (int batch = 0; batch < traindata.getNumOfBatches(); batch++)
    {
        Y = nn.forward(traindata.getBatches().at(batch));
        nn.backprop(Y, traindata.getTargets().at(batch));

        LinearLayer* linearLayer = dynamic_cast<LinearLayer*>
        (nn.getLayers()[2]);
        Matrix layerW = linearLayer->getWeightsMatrix();
        cost += crossEntropy.cost(Y, traindata.getTargets().at(batch),
        layerW);
    }
```

```
    if (epoch % 1 == 0)
        std::cout << "Epoch:" << epoch << ",Cost:" <<
        cost / traindata.getNumOfBatches() << std::endl;
}
```

5.6. Testiranje mreže

Učitajmo podatke za testiranje te ih također raspodjelimo na grupe od 32 primjera. U ovom koraku, provodimo samo prolaz unaprijed (forward pass) kroz mrežu za svaku grupu podataka. Za svaku grupu izračunavamo točnost, a na kraju ispisujemo prosječnu točnost temeljenu na svim grupama podataka.

```
MNIST testdata(32, 312, "datasets/mnist_test.csv");

    Matrix A;

    float accuracy = 0.0f;
    for (int i = 0; i < testdata.getNumOfBatches(); i++)
    {
        A = nn.forward(testdata.getBatches().at(i));
        accuracy += nn.computeAccuracy(A, testdata.getTargets().at(i));
    }

    std::cout << "Accuracy on test data: " << accuracy /
    testdata.getNumOfBatches();
```

5.7. Rezultati

Kako možemo primijetiti, ukupni trošak se smanjuje nakon svake epohe, što je pozitivan znak da mreža uspješno uči tijekom treninga. Na kraju, primjećujemo da je preciznost na testnim podacima dosegnula 96.76%.

```
Epoch: 0, Cost: 2.22482
Epoch: 1, Cost: 1.1366
Epoch: 2, Cost: 0.873096
Epoch: 3, Cost: 0.705833
Epoch: 4, Cost: 0.601317
Epoch: 5, Cost: 0.5202
Epoch: 6, Cost: 0.493883
Epoch: 7, Cost: 0.444733
Epoch: 8, Cost: 0.376507
Epoch: 9, Cost: 0.378984
Epoch: 10, Cost: 0.342434
Accuracy on test data: 0.967648
```

Ako reproduciramo istu mrežu u Pythonu i izračunamo njezinu preciznost, primjećujemo da je preciznost slična, odnosno iznosi 97.32%. U nastavku je naveden kod kako se stvara ista mreža u Pythonu.

```
import tensorflow as tf
from tensorflow.keras import layers, models

model = models.Sequential([
    layers.Flatten(input_shape=(28, 28)),
    layers.Dense(128, activation='relu'),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')])


model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])


mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels)
= mnist.load_data()

train_images, test_images = train_images / 255.0, test_images / 255.0

model.fit(train_images, train_labels, epochs = 10,
batch_size = 32, validation_data = (test_images, test_labels))

test_loss, test_acc = model.evaluate(test_images, test_labels)
print("Test_accuracy:", test_acc)
```

Literatura

[1] Charu C. Aggarwal, *Neural Networks and Deep Learning (1st ed. 2018)*, Springer

[2] `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`

[3] `https://github.com/pjreddie/darknet`

[4] `https://github.com/NVlabs/tiny-cuda-nn`

[5] R. Scitovski, *Numerička matematika*, Sveučilište Josipa Jurja Strossmayera u Osijeku – Odjel za matematiku, Osijek, 2015.