Sieć neuronowa – dokumentacja

Kamil Przybyła

30 maja 2019

1 Informacje ogólne

Niniejszy projekt jest implementacją prostej sieci neuronowej o relatywnie elastycznej architekturze, pozwalającej na dodawanie dowolnej liczby warstw z wybraną funkcją aktywacyjną. Istnieje również możliwość wyboru funkcji błędu, używanej przy ocenie skuteczności działania sieci oraz przy jej uczeniu.

Interakcja z użytkownikiem prowadzona jest przez interfejs tekstowy. Użytkownik ma możliwość tworzenia sieci o zadanej budowie, wyboru hiperparametrów oraz zapisaniu nauczonego modelu sieci do pliku, kiedy jego skuteczność jest zadowalająca. Poprawnie nauczony model może rozpoznawać cyfry znajdujące się w obrazie o rozmiarach 28x28, wskazanym przez użytkownika.

Aplikacja została napisana w taki sposób, aby jej skalowanie nie wymagało głębokich modyfikacji w kodzie. Dodanie nowego typu warstwy z inną funkcją aktywacyjną sprowadzi się do napisania klasy dziedziczącej po typie bazowym warstw i zaimplementowaniu dwóch metod: funkcji aktywacyjnej oraz jej pochodnej. Rozszerzenie sieci o nową funkcję błędu odbywa się na podobnej zasadzie.

2 Testowanie

W celu zbadania poprawności napisanego kodu sporządzono testy automatyczne jak i manualne. Testy automatyczne pokrywają kod odpowiedzialny za działanie klasy macierzy oraz poprwaność zapisywania sieci do pliku. Manualnie zaś sprawdzono czy aplikacja prawidłowo przetwarza dane zawarte w obrazie podanym przez użytkownika i czy jest w stanie rozpoznawać cyfry z takich obrazków. Wykorzystano także narzędzie valgrind do sprawdzenia czy nie występują wycieki pamięci.

3 Hierarchia klas

Najważniejszą klasą w całym projekcie jest `NeuralNetwork` reprezentująca dowolną sieć neuronową. Zawiera ona w sobie m.in. listę sprytnych wskaźników na obiekty typu `Layer` oraz wskaźnik na `CostFunctionStrategy`. Wszystkie te klasy korzystają z szablonowej klasy `Matrix`, odpowiedzialnej za reprezentację macierzy oraz wykonywanie na niej działań. Wykorzystane zostały wzorce projektowe *Stategii (Strategy)* oraz *Metody szablonowej (Template Method)*. Strategią dla sieci jest sposób, w jaki liczona jest funkcja błędu, co zaimplementowane jest we wcześniej wspomnianej klasie. Metodami szablonowymi zaś są funkcje wykonywane przez warstwy, których dokładne działanie zależy od użytej funkcji aktywacyjnej.

Za komunikację z użytkownikiem odpowiada klasa `UserInterface`, która została zrealizowana jako statyczna klasa zawierająca stany, które są zmieniane przez interakcję z użytkownikiem. Ze względu na to, iż to użytkownik kieruje działaniem aplikacji, klasa ta odpowiada za tworzenie odpowiednich obiektów, kiedy zajdzie taka potrzeba.

Wczytywanie danych trenigowych i testowych dla sieci wykonywane jest przez klasę `MNISTDataLoader`. Wspomniane dane opakowane są w klasę `MNISTData`. Za wczytywanie obrazków odpowiada klasa `Image`.

Poniżej można znaleźć deklarację interfejsów klas.

3.1 Matrix

```
template<typename T>
class Matrix
{
public:
    typedef T* iterator;
    typedef const T* const_iterator;

    Matrix(unsigned int rows, unsigned int columns);
    explicit Matrix(): Matrix(1, 1) {}
    Matrix(const T* data, unsigned int rows, unsigned int columns);
    Matrix(const Matrix& o): Matrix(o.data_, o.rows_, o.columns_) {}
    Matrix(Matrix&& o);

    ~Matrix();

    const_iterator cbegin() const { return data_; }
    const_iterator cend() const { return data_ + len_; }

    iterator begin() { return data_; }
    iterator end() { return data_ + len_; }

    unsigned int getRows() const;
    unsigned int getColumns() const;
    const T* getData() const;
    T get(unsigned int i, unsigned int j) const;
    void zero();
    void randomize(T min, T max);
    Matrix map(std::function<T(T)> const& f) const;
    T sum() const;
    Matrix hadamard(const Matrix& o) const;
    Matrix static transpose(const Matrix& m);

    Matrix& operator=(const Matrix& o);
    Matrix& operator=(Matrix&& o);
    Matrix operator+(const Matrix& o) const;
    Matrix& operator+=(const Matrix& o);
    Matrix operator-(const Matrix& o) const;
    Matrix& operator-=(const Matrix& o);
    Matrix operator*(T f) const;
    Matrix& operator*=(T f);
```

```
    Matrix operator*(const Matrix& o) const;

    friend Matrix operator*(T f, const Matrix& m);
    friend std::ostream& operator<<(std::ostream& os, const Matrix& m);
};
```

3.2 MNISTData

```
class MNISTData
{
public:
   MNISTData() = default;

    MNISTData(const MatrixVec& trainingData, const MatrixVec& trainingLabels,
              const MatrixVec& testingData, const MatrixVec& testingLabels);

    const MatrixVec& getTrainingData() const;
    void setTrainingData(const MatrixVec& trainingData);

    const MatrixVec& getTrainingLabels() const;
    void setTrainingLabels(const MatrixVec& trainingLabels);

    const MatrixVec& getTestingData() const;
    void setTestingData(const MatrixVec& testingData);

    const MatrixVec& getTestingLabels() const;
    void setTestingLabels(const MatrixVec& testingLabels);
};
```

3.3 MNISTDataLoader

```
class MNISTDataLoader
{
public:
    static MNISTData* loadData(const char* trainingImagesFilename,
                               const char* trainingLabelsFilename,
                               const char* testingImagesFilename,
                               const char* testingLabelsFilename);
};
```

3.4 UserInteraction

```
class UserInterface
{
public:
    static void handleInteraction();
};
```

3.5 Image

```
class Image
{
public:
    Image(const char* fileName);
    Image(const Image& other);
    ~Image();

    int getWidth() const;
    int getHeight() const;
    const char* getPixels() const;

    Image& operator=(const Image& o);
};
```

3.6 data_load_failure

```
class data_load_failure : public std::exception
{
public:
    data_load_failure(const char* filename, const char* additional = "")
    {
        std::stringstream ss;
        ss << "Could not load data from file named " << filename << "!" <<
            additional;
        message_ = ss.str();
    }

    const char* what() const throw()
    {
      return message_.c_str();
    }
};
```

3.7 NeuralNetwork

```
typedef float NNDataType;
typedef Matrix<NNDataType> NNMatrixType;

class NeuralNetwork
{
public:
    NeuralNetwork(unsigned int inputNodes, float learningRate,
        std::unique_ptr<CostFunctionStrategy> costFunction);

    unsigned int getLayersCount() const;

    template<typename T>
    void addLayer(unsigned int nodes);
```

```
    NNMatrixType feedforward(const NNMatrixType& input) const;

    void train(unsigned int epochs,
               unsigned int batchSize,
               const std::vector<std::shared_ptr<NNMatrixType>>& inputs,
               const std::vector<std::shared_ptr<NNMatrixType>>& targets);

    float test(const std::vector<std::shared_ptr<NNMatrixType>>& inputs,
              const std::vector<std::shared_ptr<NNMatrixType>>& targets) const;

    void save(const char* filename) const;
    static NeuralNetwork* load(const char* filename);
};
```

3.8 Layer

```
class Layer
{
friend class NeuralNetwork;
public:
    Layer(unsigned int nodes, unsigned int prevNodes);

    virtual unsigned int getNodesCount() const;

    virtual NNDataType activationFunction(NNDataType value) const = 0;
    virtual NNDataType activationDerivative(NNDataType value) const = 0;

    virtual NNMatrixType feedforward(const NNMatrixType& input, NNMatrixType&
        weightedInput);
    virtual NNMatrixType feedforward(const NNMatrixType& input) const;

    virtual NNMatrixType backpropagate(const NNMatrixType& error,
                                       const NNMatrixType& weightedInput,
                                       const NNMatrixType& prevOutput);

    virtual void performSDGStep(float learingRate);

    virtual void serialize(std::ofstream& ofile) const = 0;
};
```

3.9 SigmoidLayer

```
class SigmoidLayer : public Layer
{
friend class NeuralNetwork;
public:
    SigmoidLayer(unsigned int nodes,
        unsigned int prevNodes) :
        Layer(nodes, prevNodes) {};

    virtual NNDataType
        activationFunction(NNDataType
        value) const;
    virtual NNDataType
        activationDerivative(NNDataType
        value) const;

    virtual void
        serialize(std::ofstream&
        ofile) const;
};
```

3.10 ReLULayer

```
class ReLULayer : public Layer
{
friend class NeuralNetwork;
public:
    ReLULayer(unsigned int nodes,
        unsigned int prevNodes) :
        Layer(nodes, prevNodes) {};

    virtual NNDataType
        activationFunction(NNDataType
        value) const;
    virtual NNDataType
        activationDerivative(NNDataType
        value) const;

    virtual void
        serialize(std::ofstream&
        ofile) const;
};
```

3.11 CostFunctionStrategy

```
class CostFunctionStrategy
{
public:
    virtual NNDataType calculateCost(const NNMatrixType& output, const
        NNMatrixType& target) const = 0;
    virtual NNMatrixType calculateCostDerivative(const NNMatrixType& output,
        const NNMatrixType& target) const = 0;

    virtual void serialize(std::ofstream& ofile) const = 0;
};
```

3.12 CrossEntropyCost

```
class CrossEntropyCost : public
    CostFunctionStrategy
{
public:
    virtual NNDataType
        calculateCost(const
        NNMatrixType& output, const
        NNMatrixType& target) const;
    virtual NNMatrixType
        calculateCostDerivative(const
        NNMatrixType& output, const
        NNMatrixType& target) const;
    virtual void
        serialize(std::ofstream&
        ofile) const;
};
```

3.13 MeanSquereErrorCost

```
class MeanSquereErrorCost : public
    CostFunctionStrategy
{
public:
    virtual NNDataType
        calculateCost(const
        NNMatrixType& output, const
        NNMatrixType& target) const;
    virtual NNMatrixType
        calculateCostDerivative(const
        NNMatrixType& output, const
        NNMatrixType& target) const;
    virtual void
        serialize(std::ofstream&
        ofile) const;
};
```

4 Potencjał rozwoju

Niniejszy projekt można na wiele sposobów rozwinąć. Możliwa jest implementacja innego rodzaju warstw, jak na przykład warstwy konwolucyjne. Taki zabieg niestety prawdopodobnie wymagałby szerszych zmian w kodzie. Innym kierunkiem rozwoju jest zaimplementowanie rozszerzonych algorytmów uczenia, opierających się na *SDG*, takich jak *Momentum* czy *Adagrad*. Możliwe jest też użycie funkcji błędów z parametrem regularyzacji – nie wymagałoby to głębokich zmian w kodzie, jak i stworzenie wyspecjalizowanych sieci, takich jak *autoenkodery odszumiające/wariacyjne*.