Neural Network Implementation in the Context of a Text Based Adventure Game

Patrick Odagiu - 9914936
The University of Manchester

Object Oriented Programming in C++ - Final Project
MAY 13, 2019

A simple, three layer, neural network is used in the context of a text based adventure game. The neural network reads digits from a template image that the player interacts with, providing another input medium. The game framework consists of an abstract base class and multiple derived classes, which provide the functionality required to form a narrative. Various C++ advanced features are needed to produce the presented game including class templates, bit by bit data handling, image decoding, static variables and error handling.

I. INTRODUCTION

Machine learning algorithms [1, 2] are relatively recent computational tools whose extensive applicability and novel problem solving capacities have been gaining popularity in the past few years. Neural networks are part of this broader class of algorithms.

In the presented game, a neural network is used to facilitate an additional player input medium, diverting from the usual game-related purpose of this kind of code. The typical game scope of a neural network is exemplified by OpenAI's [3] latest work: a machine learning algorithm that managed to defeat the current *Defense of the Ancients 2* [4] world champions in an official tourney. Conversely, the neural network developed here reads digits that are drawn by the user as part of an in-game challenge and thus enriches the player experience.

The *Visual Studio 13* version of C++ 11 was used to write both the neural network and the text adventure. The game takes place in the Windows console, through which the player is compelled to make a series of difficult choices as he assumes the role of G. Freeman, an inmate presented with the opportunity to escape prison. Throughout his adventures, the player may or may not encounter various antagonists, derived from the same base Character class that includes an inventory object. Furthermore, various items are scattered throughout the game, each one derived from the same abstract base class. The player might find such items, choose to store them in his inventory and ultimately use them in his encounters with other characters. The way these encounters occur is more or less determined by the player, who must make quick decisions against a timer. The game presents five different endings, two of them worse than the other one.

The design and implementation of the neural network is outlined in Section II while the game is described in Section III. Sections IV and V give specific use cases for the code and discuss possible extensions, respectively.

II. THE NEURAL NETWORK

In conventional programming, a problem is usually solved by breaking it up into many small, precisely defined tasks that a computer can easily solve. Conversely, the programmer does not dictate a step-by-step solution to a neural network [5]; this novel type of algorithm learns from observational data and determines its own solution for a chosen problem.

A. Design

The objective is to be able to identify the single numeric handwritten digit in a 28x28 pixel mono image. Thus, a neural network with three layers and 784 (total pixels) inputs is considered. These inputs are then connected to a hidden layer of 30 neurons [5], which successively activate a final layer of 10 output neurons (one for each digit). The output neuron holding the highest activation value is returned as the recognized digit. For example, if the output neuron 1 activation is the highest, the network determined that the input image contains a handwritten 'one'.

This is achieved with a sigmoid neuron activation function and a half mean squared error cost function [5, 6]. After building the structure, the network is trained with the MNIST database of handwritten digits [7]. The MNIST data contains 60 000 training data images and 10 000 test data images. To obtain an >95% accuracy, the data is processed by the network 30 times.

The training process is optimal when choosing a learning rate of about 3.0 [5]. Furthermore, mini batches of 10 training data images are randomly chosen from the initial data set and sequentially fed to the neural network to improve the overall results [5, 6].

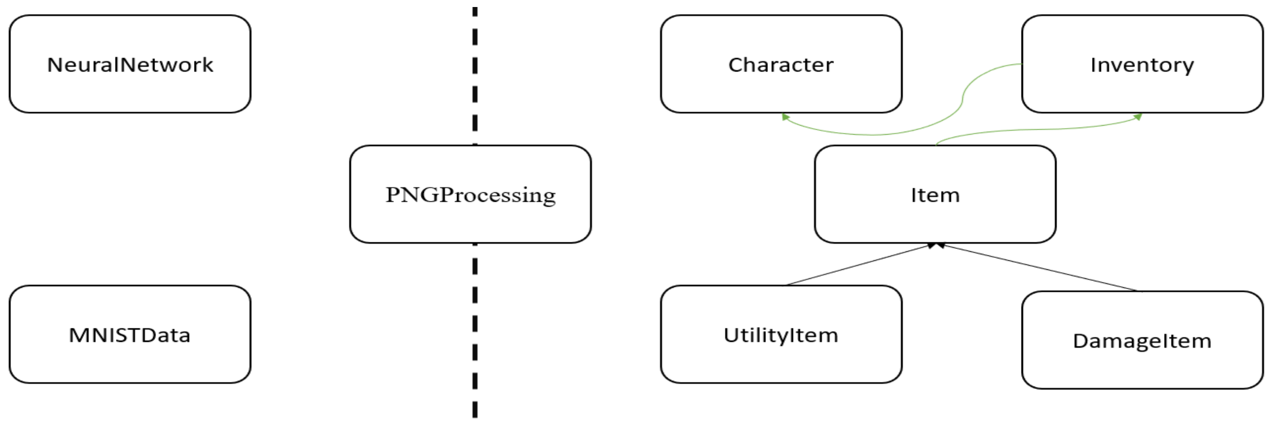

FIG. 1. The main classes. The straight lines represent an 'is a' inheritance relation. The curvy lines represent a 'has a' relation with no inheritance. The dashed line separates the neural network from the game classes.

B. Implementation

1. *Data Reading*

A class is implemented to read the training and testing data. The MNIST data comes in compressed .gz files. One training object in this data set in comprised of a label and the associated image. The format of these objects dictates that the **uint8_t** (1-byte unsigned integers spanning from 0-255) data type provides the optimal reading speed needed for the purpose of training a neural network [6].

The `MNISTData` class contains a `readImages` method, shown in Listing 1, and a similar `readLabels` method. Sometimes, the images or labels do not have the desired endianness [8] and such an `endianSwap` **inline function** is implemented and applied to process the images or the labels when needed.

```
bool MNISTData::readImages(bool train){
    const char* fileName{train ?
     ↪  "train-images.idx3-ubyte":
     ↪  "t10k-images.idx3-ubyte"};
    FILE* file = fopen(fileName, "rb");
    if(!checkOpenFile(file))
        return false;

    long fileSize{getFileSize(file)};
    imageData = new uint8_t[fileSize];
    fread(imageData, fileSize, 1, file);
    fclose(file);
    return true;
}
```

Listing 1: The `readImages` method from the MNIST data reading class, used for training the neural network.

```
inline uint32_t endianSwap (uint32_t x){
    // Manually implemented endian swap.
    return (x<<24) | ((x<<8) & 0x00ff0000) |
           ((x>>8) & 0x0000ff00) | (x>>24);
}
```

Listing 2: Edian swap inline method. Used in conjunction with the MNIST data class.

The main method of the `MNISTData` class is `load` which calls all the other methods described above and stores the image data into an `std::vector<double>` container after conversion.

2. *The Sigmoid Neural Network*

The neural network is contained in a **templated class**. The class methods are stored in a **.tpp file** that is included at the end of the class definition in a separate header file. The main methods of this class store the forward pass and backward propagation algorithms. Details of their implementation will not be given here, since no interesting C++ features are presented, but the recipe given in references [5] and [6] is followed. The template was added such that the neural network is initialised by typing `NeuralNetwork<inputNeurons, hiddenNeurons,outputNeurons> neuralNet;` thus enlarging the applicability of this particular neural network implementation. For handwritten digit recognition, one sets these to `inNeurons = 784, hiddNeurons = 30` and `outNeurons = 10`. Standard **`std::array` containers** are used to store the neuron and moreover the neuron connections data. For example, see Listing 3, where the initialisation of the connections between the input and the hidden layers is displayed.

```
std::array<double, inputNeurons * hiddenNeurons>
↪  hidLayerWeights;
```

Listing 3: Array container initialisation for the connection between the input and hidden neurons of the neural network. All variables are template variables.

The NeuralNetwork::train method in the neural network class constructs the training data batches from Section II.A and feeds each image to the network. After the neural network makes its prediction, its answer is compared with the label. The information, i.e. either correct or wrong, is propagated back through the out and hidden layers. Thus, depending on the discrepancy between the network's answer and the label, the values held in the arrays describing the connections between the neurons are adjusted accordingly.

After only one data cycle, the accuracy of the network reaches about 90%. Further iterations bring it to about 97%, where the network cannot make anymore improvements since some images are truly indistinguishable by its standards. After training, the ability of the network is tested using the 10 000 images data set designed for this purpose. If the test gives satisfying results, a method allows the export of a neural network into a text file. Furthermore, an additional method provides the import functionality, used here in the adventure game.

III. THE ADVENTURE GAME

In the short adventure game implemented here, the player takes the role of an inmate and is tasked with escaping the prison. As soon as the game starts, the door to the player's prison cell opens and a timer is started. The game must be completed in less than 300 seconds, or the application exits. The player is granted a number of choices in each level. A certain combination of choices will lead to a certain ending. There are five different endings which can be reached in various ways.

A. Design

The design of the game follows the simple text-based adventure game style. The player starts in a closed prison cell, gets an introduction to the game and is given the choice to quit or start playing. After starting the game, ASCII art is used to display a door opening and the 300 second timer starts. From this point onward, the player is given a number of choices. In the first choice tree he can simply proceed to the next level, encounter an enemy, fight a guard, acquire items from the cell or simply wait in his cell. Almost all the choices that the player can make and their consequences are mainly facilitated through three classes: the character class, the inventory class and the items class. The main class hierarchy is illustrated in Fig. 1.

The specific characters and the items chosen for the presented iteration of the text based game are derived from the aforementioned base classes. However, the code does not limit itself one story only. The framework to support the implementation of multiple narrative variations exists. For example, new characters and new items can be added to the game with minimal effort.

The neural network detailed in Section II is used in the current narrative as an additional channel of input for the player. At some point in the game, the main character arrives in front of a locker with a code. One of the ways to open the locker is by going into the game folder and writing three numbers on a provided template image. After the player confirms this action, the neural network reads the image, identifies the digits and sets them as the code for the locker.

B. Implementation

1. The Items and Inventory

An abstract base Item class was implemented, with two derived classes: the UtilityItem class and the DamageItem class, where most of the function overloading is done. The main functionality provided here is through the attack and pryOpen methods, the former being used in combat situations and the latter being used to break through environment objects. Listing 4 shows the attack method overload for the damage item type. A **random number generator** is used to determine if an attack is successful or not.

```
size_t DamageItem::attack(double threshold){
    double roll{generateRandomNumber()};
    if(roll > threshold) return damageDealt;
    return 0;
}
```

Listing 4: The attack method for the damage item type. The threshold can be a character's agility attribute.

All the items in the game are derived from the two item type classes. This last set of classes contain the ASCII art of the item and a few other relevant attributes like the damage. One can overload some methods again in these final derived classes to provide additional functionality if an item is deemed to have some special characteristics.

```
virtual DamageItem* Clone() const
    { return new DamageItem(*this); }
```

Listing 5: Methodf from the `Item` class used in the `Inventory` class to pick up items.

A class wrapped around an `std::map` container is used to provide the inventory functionality. The two main methods of this class are `pickItem` and `dropItem`, both being designed to work for any kind of item derived form the `Item` abstract base class. To pick an item up and store it in the inventory, Listing 5 and Listing 6 are used such that if an item is created and goes out of scope, the inventory will not get affected.

```
bool Inventory::pickItem(Item* itm){
    bool fullInventory(items.size() >= 2);
    if(fullInventory)
        return false;
    else items[itm->getName()] = itm->Clone();
    return true;
}
```

Listing 6: Method through which items are picked up and stored in the inventory.

2. *The Characters*

A base `Character` class was implemented to facilitate character functionality throughout the game. Every character presented in the current iteration of the game is either directly constructed through the base class or a separate, derived class if additional overloading is required. Characters have ASCII art, dialogue lines, health and agility, as well as an public inventory object. The main functionality of the character class is exemplified through the `dealDamage` method presented in Listing 7.

```
int Character::dealDamage(Item* atkItm,
 ↪  Character* attackedChar){

if(chrInv.checkForItm(atkItm->getName())){
double chrThresh{attackedChar->getCharAgi()};
size_t dmgDealt{atkItm->attack(charThresh)};
return attackedChar->takeDamage(dmgDealt);
}
return -1;
}
```

Listing 7: Method through which characters deal damage. It returns 1 if the item killed the attacked character, 0 if damage was dealt and -1 if no damage was dealt.

3. *The Display*

All the display functionality is contained in the windowsDisplay.cpp file. This was separated from the main game classes and files since it is platform specific and required the windows.h header file to be included. The main method in this file is `placeCursor`, shown in Listing 8. This method facilitates clearing the screen and centering the text in the console. Additionally, a method which enables the full screen console is implemented.

```
void placeCursor(const size_t x, const size_t y){
    HANDLE consoleHandle{GetStdHandle(STD_OUTPUT_
     ↪  HANDLE)};
    COORD placeCursorHere;
    placeCursorHere.X = x;
    placeCursorHere.Y = y;
    SetConsoleCursorPosition(consoleHandle,
     ↪  placeCursorHere);
}
```

Listing 8: Places the console cursor at a point with coordinates x and y.

4. *Image Processing*

A class is implemented to process .png format images. This class uses the **lodepng** [9] include for the decoding. The constructor for this class is shown in Listing 9. The grey scale image produced in this class is readable by the neural network described in Section II. A **deep copy constructor, move constructor, copy assignment operator and move assignment operator** were also implemented for completeness.

```
PNGImage::PNGImage(const char* filename){
unsigned error = lodepng::decode(image, width,
 ↪  height, filename);
if(error) std::cout<<"Decoder error"<<error<<":"
 ↪  <<lodepng_error_text(error)<<std::endl;

uint32_t r,g,b; double greyPixel;
for(size_t i=0; i<image.size(); i=i+4){
    r = image[i]; g = image[i+1]; b = image[i+2];
    greyPixel = 255 - (r+g+b)/3.0;
    imageGreyScale.push_back(greyPixel/255.0);
    }
}
```

Listing 9: Constructor for importing and decoding a .png image. A greyscale image is created as well.

5. *The Timer*

The timer runs in a **parallel thread** with the rest of the code and counts down from 300 seconds. Furthermore, the display functions are used to print every tick in the top left corner and return the cursor to its original position. This timer is started once a playthrough begins and quits the game when it reaches 0 seconds.

```
void timer(){
int xCoord, yCoord;
for (int i = 0; i < 300; i++){
    xCoord = consoleX(); yCoord = consoleY();
    placeCursor(0, 0);
    std::cout << 300-i<<"     ";
    placeCursor(xCoord, yCoord);
    std::this_thread::sleep_for(std::chrono::seco⌋
     ↪  nds(1));
}
exit(1);
}
```

Listing 10: The timer method, counting down from 300 seconds. When the loop finishes, the code quits.

IV. EXEMPLIFIED FEATURES

The framework described in Section II and Section III allows for building a number of interesting text based adventure games. In the Levels.cpp and Main.cpp files, one iteration of such a game is presented.

```
std::string lockerCode(){
loadNeuralNet();
PNGImage img("img.jpg");
int digitOne{NN.forwardPass(img.getGreyscale
 ↪  Pointer(0))};
int digitTwo{NN.forwardPass(img.getGreyscale
 ↪  Pointer(32))};
int digitThr{NN.forwardPass(img.getGreyscale
 ↪  Pointer(63))};
std::ostringstream code;
code<<digitOne<<digitTwo<<digitThr;
return code.str();
}
```

Listing 11: Neural network is loaded and an image containing three digits is imported. The three digits are identified sequentially by the neural network and stored.

The neural network is used to set a lock code at some point in the game. The Listing 11 code is implemented to facilitate this action, as described previously in the previously mentioned sections. Furthermore, multiple character interactions can be established. For example, Listing 12 illustrates such an interaction, where a mechanism through which an angry guard strikes back at the player is implemented. The guard can either miss the player, deal some damage or kill the player.

```
void guardRiposte(Character* guard, Character*
 ↪  player, size_t angry){
if (angry >= 3){
displayCentredText("The guard shoots at you!", 7,
 ↪  false);
int guardAtk{guard->dealDamage(guard->
 ↪  charInv["Pistol"],player)};
if (guardAtk == 1)
displayCentredText("And kills you!", 8, false);
else if (guardAtk == 0)
displayCentredText("And hits you!", 8, false);
else
displayCentredText("And misses you!", 8, false);
}
}
```

Listing 12: Method through which the guard strikes back. For context, the guard in question gets angry when the player strikes him more than 3 times.

Finally, **static** variables are used to initialise the characters in the main game, such that their scope is restricted. Error handling is done throughout the code using multiple methods, including the **stdexcept** header.

V. FUTURE WORK AND CONCLUSION

The current code could be extended in multiple ways. One approach would be to design and tell another story with the available framework. Another approach would focus on the neural network. At this moment, its recognition power is limited since it can only read digits which are provided in a particular format. However, this can be extended to recognising fruits, animals and so on with a certain amount of effort [5]. A final extension idea would be to introduce further character interaction methods and more types of items into the adventure game. Thiswould require writing more methods for the available classes or adding more derived classes.

The adventure game, in its current iteration, together with the neural network provides a framework for bulding narratives. The presently provided example makes use of all the current functionality. However, various other scenarios could be constructed out of the provided classes.

[1] Jerome Friedman et al. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[2] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2009.

[3] Greg Brockman et al. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[4] Jose Maria Fernandez and Tobias Mahlmann. The dota 2 bot competition. *IEEE Transactions on Games*, 2018.

[5] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, USA:, 2015.

[6] Neural network recipe: recognize handwritten digits with 95% accuracy. `https://bit.ly/2VMttm2`. Accessed: 10.05.2019.

[7] Li Deng. The MNIST database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[8] Danny Cohen. On holy wars and a plea for peace. *Computer*, 14(10):48–54, 1981.

[9] Lode Vendevenne. Lodepng image decoder. `https://lodev.org/lodepng/`. Accessed: 10.05.2019.

This document contain 2483 words in text (excluding listings, captions, references etc.).