сжатия, который преобразует значения выходного вектора капсул таким образом, что меняется только его длина, а углы остаются постоянными. Таким образом, получается вектор в диапазоне значений от 0 до 1, что соответствует вероятности. Следующий шаг — это применение алгоритма динамической маршрутизации. В CNN используется операция *max pooling* как способ уменьшить размер. Однако в капсульных сетях используется другой метод — направление по соглашению (*routing by agreement*).

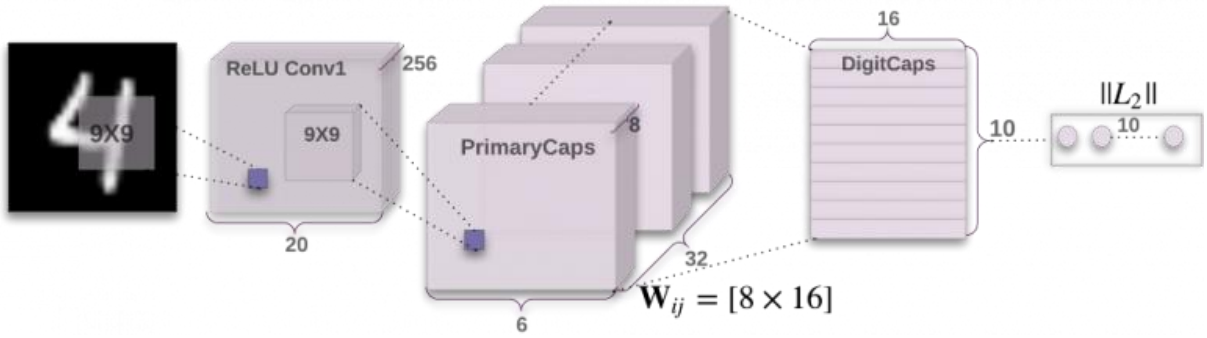

Рисунок 5 – Архитектура капсульной нейронной сети

Капсульные нейронные сети — крайне перспективная архитектура нейронных сетей для решения задачи классификации, которая улучшает распознавание изображений при изменяющихся ракурсах и иерархической структурой. Процесс обучения капсульных нейронных сетей осуществляется с помощью динамической маршрутизации между капсулами. Капсульные сети снижают ошибку распознавания объекта в другом ракурсе в сравнении со сверточными неросетями [4,5]. В настоящее время данная архитектура модифицируется и дорабатывается с целью получения результатов на сравнительно меньшем объеме данных, чем этого требуют CNN [6, 7].

Список использованных источников:

1. Сурдин, В.Г. Галактики / В.Г. Сурдин. – М.: ФИЗМАТЛИТ, 2013. – 432 с.
2. Willet, K. W. Galaxy Zoo 2: detailed morphological classifications for 304,122 galaxies from the Sloan Digital Sky Survey / K. W. Willet, Chris J. Lintott, Steven P. Bamford. – USA : Coornell University, 2013. – 30 p. – Mode of access: https://arxiv.org/pdf/1308.3496.pdf. – Date of access: 22.03.2019.
3. Sabour, S. Dynamic Routing Between Capsules / S. Sabour, N. Frosst, G. E. Hinton. – Toronto: Google Brain, 2017. – 11 p. – Mode of access: https://arxiv.org/pdf/1710.09829.pdf. – Date of access: 22.03.2019.
4. Mukhometzianov, R. CapsNet comparative performance evaluation for image classification / R. Mukhometzianov, J. Carrillo. – Canada: University of Waterloo, 2018. – 14 p. – Mode of access: https://arxiv.org/ftp/arxiv/papers/1805/1805.11195.pdf. – Date of access: 22.03.2019.
5. Xi, Ed. Capsule Network Performance on Complex Data / Ed. Xi, S. Bing, Y. Jin. – Pittsburgh: Carnegie Mellon University, 2017. – 7 p. – Mode of access: https://arxiv.org/pdf/1712.03480.pdf. – Date of access: 22.03.2019.
6. Hinton, G. E. Matrix capsules with EM routing / G. E. Hinton, S. Sabour, N. Frosst. – Toronto: Google Brain, 2018. – 15 p. – Mode of access: https://openreview.net/pdf?id=HJWLfGWRb. – Date of access: 22.03.2019.
7. Lin, A. On Learning and Learned Representation with Dynamic Routing in Capsule Networks / A. Lin, J. Li, Z. Ma. – Guangdong: Guangdong Polytechnic Normal University, 2018. – 12 p. – Mode of access: https://arxiv.org/pdf/1810.04041.pdf. – Date of access: 22.03.2019.

ЗНАЕТЕ ЛИ ВЫ, ЧТО ПРОИСХОДИТ, КОГДА ГОВОРИТЕ МИРУ “ПРИВЕТ!”?

Борушко А.Н.

Белорусский государственный университет информатики и радиоэлектроники
г. Минск, Республика Беларусь

Киклевич У.С. – ассистент кафедры ЭВМ

В данной статье на примере простейшей программы показывается, что в программировании за простыми вещами стоят более сложнейшие абстракции.

При нынешних скоростях развития языков программирования, появления новых техник, фреймворков и технологий мы начинаем забывать то, с чего все начинали.

Каждый, кто хоть когда-нибудь пробовал написать свою первую строчку кода, писал именно “Hello, World!”.

Все знают, что это такое и что мы получим в консоли, но далеко не каждый знает то, как работает программа “Привет, мир!”

Когда люди пишут такие незамысловатые программы, они даже не подозревают, что стоит за несколькими строками кода.

На примере языка программирования Си и операционной системы Windows я более подробно рассмотрю принцип работы простейшей программы.

Первое, что происходит в процессе компиляции – над программой начинает работать препроцессор. Он буквально копирует в наш main.c файл все, что задано с помощью **#include**. Таким же образом мы подключаем к нашей программе библиотеку **stdio.h**. Препроцессор же копирует в нашу программу все стандартные функции, которые объявлены в заголовочном файле **standard input/output**.

После чего компилятор начнет переводить наш код в машинный.

Компоновщику (**linker**) передается параметр **/subsystem**, который задает тип приложения, например **/SUBSYSTEM:WINDOWS** для оконных программ, или **/SUBSYSTEM:CONSOLE** для консольных программ. Это нужно операционной системе. По типу приложения, который прописывается в заголовке **.exe** файла, операционная система понимает, какое приложение ей нужно запускать - оконное или консольное.

На строчке **int main** мы обозначаем входную точку(**entry point**) для нашей программы.

На самом деле точкой входа является некоторая функция, которая будет вызвана операционной системой при старте приложения. В C/C++ это функция рантайма (**CRT**), которая проводит инициализацию **CRT**, вызывает конструкторы глобальных объектов. Для каждой платформы эта функция имеет свое название. Для Windows это **mainCRTStartup.** Если же этой стартовой точки не будет у вас программе, то компилятор выдаст ошибку:

"**Entry point must be defined**".

Стартовые точки могут быть абсолютно разными. По умолчанию стартовой точкой является функция **int main**, которая возвращает число родительскому процессу. В нашем случае родительским процессом является терминал. Терминалом определено, что дочерние процессы должны возвращать 8-ми битные целые числа, которые служат индикатором успешного завершения программы или завершения с ошибкой.

Внутри этой функции мы определяем поведение программы. В нашем случае мы вызываем функцию **printf** из стандартной библиотеки, которая может выводить данные разных типов в терминал. В конце мы возвращаем число ноль родительскому процессу, чтобы обозначить успешное завершение программы.

Стандартный компилятор Windows позволяет скомпилировать Си код, но не проводить компоновку, что дает нам возможность просмотреть **.obj** файл нашей программы. В нем хранится машинный код программы. В заголовке объектного файла имеются три основных сегмента: .**data, .drectve, .text$mn.**

В секции .**data** хранятся инициализированные глобальные и статические поля.

В **.drectve** хранятся команды для компоновщика. Данная секция удаляется после компоновки объектного файла.

.text$mn или **text segment** содержит в себе скомпилированные под платформу X машинные инструкции. Во время выполнения **.exe** файла загрузчик программ наполнит определенный сектор процесса нашей программы этими инструкциями.

После компиляции генерируется символьная таблица и таблица перенаправлений.

Символьная таблица (Рисунок 1) содержит всю информацию по использованным символам, на которые ссылается подаваемый на вход компилятора исходный текст.

В первой колонке таблицы содержится индекс символа или его номер. Во второй виден адрес символа перед перемещением.

Если в третьей колонке содержится **SECT** и номер – это означает, что символ определен в данной секции объектного файла, если **UNDEF**, то символ находится в глубине **CRT**. Также в символьной таблице можно увидеть является ли символ **External** или **Static.** Если он **Static**, то символ виден только в исходном тексте нашей программы (**translation unit**), если **External,** то он виден всем остальным файлам.

В таблице перенаправлений, в колонке под названием Offset, указывается адрес, куда мы перенаправляем символ. А в колонке Applied To указан адрес символа в той или иной секции объектного файла.

С помощью системной утилиты **dumpbin** мы можем просмотреть assembly код. Так как assembly имеет одинаковые маппинги с машинным кодом, то в нем есть: адрес памяти, машинный код и эквивалентные ему команды assembly. В нем содержится функция **_main,** которая делает вызов **_printf.** Сам **_printf** делает еще два вызова, один прямиком в **CRT** чтобы добавить возможность стандартного ввода/вывода, и **_vfprintf_l**, которая используется для работы самой функции **printf**. В версии компилятора начиная с 2015 года, части функции **printf** уже содержатся в заголовочном файле.

Для того, чтобы получить **.exe** файл, нужно провести компоновку объектного файла. В заголовке **.exe** файла можно увидеть дополнительную секцию **.rdata.** Именно в этой секции находится строчка со значением “Hello, World!”

Таким образом я продемонстрировал, что в программировании ничего не работает просто так. Даже за самой минимальной программой лежит большое количество “прослоек”, которые работают не идеально и требуют постоянных доработок и оптимизации.

```
Dump of file Source.obj

File Type: COFF OBJECT

COFF SYMBOL TABLE
000 01046992 ABS    notype       Static       | @comp.id
001 80000191 ABS    notype       Static       | @feat.00
002 00000000 SECT1  notype       Static       | .drectve
    Section length   2F, #relocs    0, #linenums    0, checksum        0
004 00000000 SECT2  notype       Static       | .debug$S
    Section length   78, #relocs    0, #linenums    0, checksum        0
006 00000000 SECT3  notype       Static       | .text$mn
    Section length   14, #relocs    2, #linenums    0, checksum F829D91C
008 00000000 SECT4  notype       Static       | .text$mn
    Section length    A, #relocs    1, #linenums    0, checksum 71A05264, selection    2 (pick any)
00A 00000000 SECT5  notype       Static       | .text$mn
    Section length   29, #relocs    2, #linenums    0, checksum 2B25B17F, selection    2 (pick any)
00C 00000000 SECT6  notype       Static       | .text$mn
    Section length   3A, #relocs    2, #linenums    0, checksum CAE6D625, selection    2 (pick any)
00E 00000000 SECT4  notype ()    External     | ___local_stdio_printf_options
00F 00000000 UNDEF  notype ()    External     | ___acrt_iob_func
010 00000000 UNDEF  notype ()    External     | ___stdio_common_vfprintf
011 00000000 SECT5  notype ()    External     | __vfprintf_l
012 00000000 SECT6  notype ()    External     | _printf
013 00000000 SECT3  notype ()    External     | _main
014 00000008 UNDEF  notype       External     | ?_OptionsStorage@?1??__local_stdio_printf_options@@9@9
rintf_options'::`2'::_OptionsStorage)
015 00000000 SECT7  notype       Static       | .data
    Section length    D, #relocs    0, #linenums    0, checksum AF57950C
017 00000000 SECT7  notype       Static       | $SG7440
018 00000000 SECT8  notype       Static       | .chks64
    Section length   40, #relocs    0, #linenums    0, checksum        0
```

Рисунок 1 – Символьная таблица

ВЫЯВЛЕНИЕ DNS ТУННЕЛИРОВАНИЯ В КОМПЬЮТЕРНОЙ СЕТИ МЕТОДОМ POMDP

Бубнов Я.В.

Белорусский государственный университет информатики и радиоэлектроники
г. Минск, Республика Беларусь

Иванов Н.Н – к.физ-м.н.,доцент

В статье предлагается способ выявления целевой кибератаки, в частности DNS туннелирования в компьютерной сети. Предложенный метод базируется на использовании параметризированного частично-наблюдаемого марковского процесса принятия решений.

DNS туннелирование — это техника инкапсуляции произвольных данных через протокол системы доменных имен. В течение последних лет атаки, связанные с кражей информации из платежных систем, так или иначе эксплуатировали данную уязвимость протокола. Ярким примером может служить программа FrameworkPOS [1], с помощью которой в 2014-2015 годах осуществлялась масштабная кража информации о кредитных картах с платежных систем. Основная трудность, связанная с выявлением DNS туннелирования, заключается в сложности дифференциации туннелируемого трафика от обычного.

В виду того, что отключение системы DNS в компьютерных сетях представляется принципиально невозможным, требуются более интеллектуальные способы решения задачи – обнаружения DNS туннелирования.

За последние годы исследования проблемы обнаружения DNS туннелирования в компьютерных сетях предложено некоторое количество подходов в построении детекторов. Например, в статье [2] Надлер, Аминов и Шабтай решают проблему обнаружения туннелирования