High-Performance Computer Architectures
Practical Course

Script

High-Performance Computing
Prof. Dr. Ivan Kisel
Robin Lakos
Akhil Mithran
Oddharak Tyagi

June 3, 2025

Dear students,

Welcome to the High-Performance Computer Architectures practical course. The goal of this course is to provide you an introduction to software development for high-performance computers including simplified applications from our daily research routine. The modern low-level programming language C++ allows to develop software close to the hardware specifications and allows to program runtime- and memory-efficient applications, running on CPU and GPU. Within this course, you will learn how to work with Unix-based systems and C++ to write smaller applications that make use of several concepts of parallel programming such as multi-threading and vectorization.

If you have any questions regarding the examples given, please ask your tutor.

Please note: During the course, we will add new sections and provide more examples for the specific topics to provide a good overview of the course requirements. If you have any suggestions or wishes that could help you, please let us know.

Contents

Chapter 1

Introduction - Unix Shell and C++

The Unix Shell and C++ are essential tools for anyone working in high-performance computing or scientific research. Both provide robust capabilities for managing and executing tasks efficiently. This chapter is divided into two sections: the first introduces the Unix Shell as a versatile command-line interface for interacting with operating systems, and the second provides an introduction to C++ programming, emphasizing its role in computational science.

1.1 Unix Shell

The Unix Shell acts as a powerful interface between users and the operating system, allowing direct interaction through text-based commands. Its flexibility and precision make it an indispensable tool for professionals, particularly in high-performance computing, where efficiency and automation are crucial. Unlike graphical user interfaces (GUIs), which prioritize user-friendliness, the shell excels in handling complex, repetitive, and large-scale tasks with ease, thanks to its focus on command-line execution and scripting.
Unix, developed in 1969, laid the foundation for many modern operating systems, including Linux and macOS. Its principles of multi-tasking, multi-user support, and portability have made it a cornerstone of computing systems worldwide. Today, Linux, a Unix-based kernel, powers a vast array of devices, from personal computers to high-performance computing (HPC) clusters and servers.

Linux's open-source nature is one of its greatest strengths ind HPC and research. It allows developers and researchers to inspect, modify, and optimize the source code to suit specific needs. This flexibility, coupled with contributions from a global community, ensures robust performance, rapid issue resolution, and high adaptability for demanding computational tasks.

Linux distributions come in many flavors, tailored to different use cases. For instance, Debian and Fedora are often chosen for servers due to their stability and reliability. Conversely, distributions like Ubuntu and Arch Linux cater to desktop users with user-friendly interfaces and up-to-date software. The availability of diverse Desktop Environments (DEs) further allows users to customize their workflows, enhancing productivity and accessibility.
In the realm of HPC, Linux dominates as the operating system of choice. Its stability, scalability, and extensive support for open-source tools and frameworks make it indispensable for running large-scale simulations, managing big data, and optimizing computational workflows.

1.1.1 Terminal and Shell Commands

In this course, all C++ applications will be compiled and executed via the terminal (also referred to as the console), using the Unix Shell. Working through the terminal allows precise control and avoids the overhead introduced by graphical user interfaces or additional software. While initially it may feel less intuitive, the terminal is a powerful tool for efficient computing, particularly in high-performance computing environments.

When you start a terminal in Linux, the prompt typically displays `username@hostname`, followed by a *path*, which represents the current *working directory*. By default, the working directory is usually set to the user's home directory. Note that terminal prompts and styles may vary depending on the application or theme being used.

Key Commands for Navigation
The following table summarizes essential commands for navigating the file system in the Unix Shell:

Command	Description
`pwd`	Prints the working directory, showing the absolute path to the current location.
`cd`	Changes directory. Paths can be absolute (starting from the root directory, `/`) or relative (from the current directory).
`ls`	Lists the contents of the current directory.

Table 1.1: Essential Unix Shell commands for navigation.

Below is an example of using these commands in a typical workflow:

```
username@hostname:~$ pwd
/home/hpc
username@hostname:~$ ls
directory-A directory-B directory-C
username@hostname:~$ cd directory-A
username@hostname:~/directory-A$ ls
directory-1 directory-2
username@hostname:~/directory-A$ cd directory-1/directory-1-1
username@hostname:~/directory-A/directory-1/directory-1-1$ cd ../../..
username@hostname:~$
```

Switching between directories with long paths can become tedious. The command **cd -** offers a convenient shortcut, allowing you to quickly switch back to the last directory you were working in. This is especially helpful when toggling between two separate locations:

```
username@hostname:~$ pwd
/home/hpc
username@hostname:~$ cd directoryA/directory-1/directory-1-1
username@hostname:~/directoryA/directory-1/directory-1-1$ DO SOMETHING HERE
username@hostname:~/directoryA/directory-1/directory-1-1$ cd
↪  ../../../directoryB/directory-3/directory-2-1
```

```
username@hostname:~/directoryB/directory-3/directory-2-1$ DO SOMETHING THERE
username@hostname:~/directoryB/directory-3/directory-2-1$ cd -
username@hostname:~/directoryA/directory-1/directory-1-1$ DO SOMETHING HERE
↪ AGAIN
```

Mastering the shell is essential for anyone aiming to work efficiently in high-performance computing. Throughout this course, make an effort to familiarize yourself with the terminal, as it will be a key tool for compiling, running, and managing your C++ applications.

1.2 C++ Basics

C++ is one of the most widely used programming languages in scientific computing and high-performance applications. Known for its efficiency, flexibility, and extensive control over system resources, C++ combines the power of low-level hardware access with modern high-level abstractions. These features make it an indispensable tool in fields such as computational physics, engineering simulations, and data-intensive research. In this section, we introduce the foundational concepts of C++ and its relevance to modern programming.

1.2.1 The History and Evolution of C++

C++ was first developed by Bjarne Stroustrup in 1985 as an extension of the C language. The goal was to combine the performance and portability of C with features supporting object-oriented programming, such as classes and objects. This key enhancement distinguished C++ as one of the first mainstream programming languages to support object-oriented design, alongside procedural and generic programming paradigms. C++ has grown significantly since its inception, evolving through multiple standards established by the International Organization for Standardization (ISO). Starting from C++98, these standards have introduced features like templates, smart pointers, and lambda expressions, with major updates occurring every three years since 2011. This makes C++ a ”modern” language that balances its low-level control with high-level abstractions. While C++ is often considered a high-level language due to its expressive syntax, it retains low-level features like direct memory management and hardware-level operations. This versatility positions C++ between low-level languages like Assembly and modern high-level languages like Python, making it highly efficient and performant, particularly for computationally intensive tasks.

1.2.2 Learning Resources for C++

Given the breadth of the language, this course will focus on essential concepts and practices relevant to high-performance computing. For more comprehensive learning, the following resources are highly recommended:

Online resources:

- https://cppreference.com/
  A precise and extensive reference for C++ language features and libraries, complete with examples.
- https://isocpp.github.io/CppCoreGuidelines/
  A guide to writing efficient, maintainable, and safe C++ code.

1.2.3 Repetition: Programming Languages

Before diving into C++, this section revisits some foundational concepts of programming to ensure a clear understanding of key terms and ideas.

Definitions:

- A *computer* is a device that executes programs.
- A *program* is a precise sequence of instructions, enabling a computer to perform a specific task.

How Computers Understand Instructions

At the core of programming is the interaction between a computer and the instructions it executes. To fully appreciate the role of programming languages like C++, it's crucial to understand how computers interpret and process instructions at the most fundamental level.

A computer fundamentally "understands" binary, as its electronic components operate based on two states: voltage present (1) or not present (0). This binary logic is foundational to digital computing, where thresholds are applied to distinguish between these states, ensuring resilience against fluctuations and noise. Boolean logic is then used to process these signals efficiently.

The exact binary instructions a computer understands depend on its hardware architecture. For instance, ARM-based processors interpret instructions differently than x86_64-based processors. Data is represented in binary, too, but its interpretation depends on the specified *data type*. Consider the following binary sequence:

```
01100101 01110110 01101001 01101100
```

- interpreted as a 32-bit float (IEEE-754) results in `7.272793e22`.
- interpreted as a 32-bit integer results in `1702259052`.
- interpreted as ASCII/UTF-8 text results in `"evil"`.

Understanding how computers work with binary at this level forms the foundation for learning how programming languages bridge the gap between human-readable instructions and machine-readable operations.

While computers operate on binary instructions, it would be impractical and time-consuming for humans to write programs directly in binary. High-level programming languages were developed to simplify this process, allowing humans to write instructions that are closer to natural language while maintaining the precision required for computers.

High-level programming languages provide an abstraction layer that shields programmers from the complexity of binary or low-level machine instructions. Unlike human languages, which allow for interpretative flexibility, programming languages are strictly defined with precise syntax

and semantics. Each statement corresponds to a specific operation defined by the language's rules.

Since computers only "speak" binary, instructions written in high-level languages need to be translated into machine-readable binary instructions. This translation is typically performed by a *compiler* or an *interpreter*:

- A *compiler* translates the entire source code into a binary executable file at once. This executable can then be run repeatedly without needing the source code, as long as the target computer uses the same architecture.
- An *interpreter*, on the other hand, translates and executes source code line-by-line at runtime, without producing a standalone executable.

This distinction between compilers and interpreters highlights a key difference in how programming languages are implemented, which is particularly relevant when considering C++.

C++ as a Compiled Language

C++ is a high-level programming language that is almost[1] always implemented as a compiled language. This means that programs written in C++ are first translated into binary executable files by a compiler before they can be executed.

This compilation process gives C++ its reputation for efficiency and speed, as the resulting binary is optimized for the specific architecture on which it will run. Numerous vendors provide C++ compilers, including the Free Software Foundation, LLVM, Microsoft, Intel, Embarcadero, Oracle, and IBM. For this course, the recommended compilers are:

- `g++` (GNU Compiler Collection) by the Free Software Foundation.
- `clang++` by LLVM.

By understanding the role of compilers and how they relate to C++, you'll be better equipped to write efficient programs and manage their execution across different systems.

1.2.4 Compiling C++ Source Code - Single Source File

As already mentioned, the compilation of source code is the translation from a programming language into machine instructions. For a single file, this process is more intuitive, so we begin with a simple example. Later, we will explore more complex projects in greater detail.

The Minimal C++ Program

Before diving into a full-fledged example, let's create the simplest possible C++ program, containing only an empty `main()` function. This serves as a starting point to demonstrate how to compile and execute C++ code.

Create a plain text file with the `.cpp` extension, named `Minimal.cpp`, and add the following content:

[1] Some Just-In-Time (JIT) compilers, such as the one included in the ROOT Framework developed at CERN, enable C++ to behave somewhat like an interpreted language.

```cpp
int main()
{
    return 0;
}
```

Explanation:

- The `main()` function is the entry point of every C++ program. When the program is executed, this function is called first.
- The `return 0;` statement signals to the operating system that the program has executed successfully. Returning a non-zero value typically indicates an error.

To compile and execute this minimal program:

1. Compile the source code into an executable file using the `clang++` compiler:
   `clang++ Minimal.cpp -o Minimal.out`
2. Run the resulting executable:
   `./Minimal.out`
3. Since this program has no output, nothing will appear in the console.

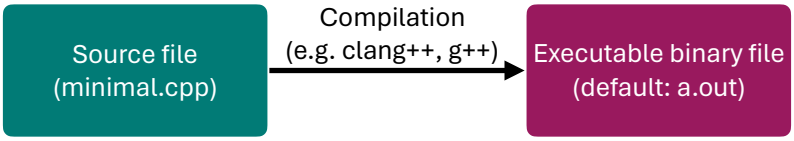

Figure 1.1: Translation of C++ source code into executable machine instructions.

This minimal example demonstrates the steps to compile and run a C++ program.

The ”Hello World!” Program

Now, let's expand on this by creating a more illustrative program. The ”Hello World!” example introduces basic input and output functionality in C++. Start by creating another file, named `HelloWorld.cpp`, and add the following content:

```cpp
#include <iostream>

int main()
{
    std::cout « ”Hello world!” « std::endl;
    return 0;
}
```

Explanation:

- The `#include <iostream>` directive imports the `iostream` library, enabling input and output operations.

- `std::cout` is used to output text to the console. The « operator streams the string `"Hello world!"` to the standard output.
- `std::endl` adds a newline character and flushes the output buffer, ensuring the output appears immediately.
- The program ends with `return 0;`, signaling successful execution.

To compile and execute this program:

1. Compile the source code:
   `clang++ HelloWorld.cpp -o HelloWorld.out`
2. Run the executable:
   `./HelloWorld.out`

These two examples - `Minimal.cpp` and `HelloWorld.cpp` - provide a foundational understanding of compiling and running C++ programs.

1.2.5 Primitive/Fundamental Data Types in C++

There are several *primitive* or *fundamental data types* in C++, such as those listed in Table 1.2, that help programmers to build more complex structures and data types.

Data Type	(Typical) Size in Bits
`bool`	1
`char`	8
`int`	32
`long`	64
`float`	32
`double`	64

Table 1.2: Example of a few fundamental data types and their typical sizes on modern PCs.

Here, these data types are given with their typical sizes in bytes. In the C++ standard, sizes of fundamental data types are given as minimum required precision, not as a fixed size value. For example, the data type `int` requires at least 16 Bit / 2 Bytes, but usually comes with 32 Bit / 4 Bytes on modern computer architectures. More on that here: https://en.cppreference.com/w/cpp/language/types.

This decision is rooted historically and is influenced by the need for portability and compatibility (e.g. backward compatibility) across various platforms, including those with different word sizes and memory constraints, e.g. in embedded systems.
These sizes are sometimes neglected as long as the used data types are sufficiently large to store the data. In high-performance computing, however, these can have crucial factors with respect to the runtime. For example, the charge of a particle in heavy-ion physics experiments can have one of three different states: positively (+1), neutrally (0) or negatively (-1) charged. Therefore, a datatype that can store at least three different states is sufficient.
A bool with only two states (true, false) is not sufficient and the next largest integer type that can represent it is a `short` with 16 Bit on modern computers. Intuitively, this would make sense as it

can easily store negative and positive numbers. However, a `char` has a size of 8 Bits and can also easily represent the three states. Although it is a character type, it can be used to store the charge of a particle. For each particle, there are 2 times less Bits required to store its electrical charge when using `char` in comparison to `short`.

But does it make a difference? Well, it depends. If the computer reads a line of aligned memory (RAM) and stores it into the cache, twice as much values can be stored when we would store the charges of particles as `char`. Access to the memory is quite costly, that sometimes even re-calculation can be much faster than reading it from memory. Just to provide some numbers: a memory access can easily take up to 100 ns, whereas L1 cache access can be as fast as 0.5 ns. Nevertheless, to achieve an efficient runtime, optimal data types help definitely, but how they are stored and aligned in memory is at least as crucial as finding the best matching one. Later in this course, different types of storing data are discussed in more detail.

1.2.6 Types and Casting

C++ is a statically and strongly typed language, which means the types of variables must be known at compile-time. This attribute enforces type safety by generating errors if variable types do not match during compilation, limiting implicit conversions. It is imperative in C++ that every variable and function return type is explicitly declared. In C++ 11, the *auto* keyword was introduced, enabling the compiler to infer variable types at compile time, enhancing code readability and maintainability, especially in complex scenarios like iterating over containers or using lambdas.

Regarding type casting, C++ supports conversions between different types, categorized into implicit and explicit conversions. Implicit conversions are automatically performed by the compiler, whereas explicit conversions require specific syntax to instruct the compiler. Although C++ maintains backward compatibility with C-style casting, it is advisable to utilize C++'s four cast operators for clearer, type-safe conversions. This recommendation stems from the fact that C-style casting can allow conversions between incompatible types, posing risks for code maintenance and clarity, especially in collaborative environments.

The C++ cast operators are:

- `static_cast<T>`: Suitable for non-polymorphic type conversions, including upcasting and downcasting within class hierarchies, numeric type conversions, and pointer type conversions where implicit or reversible. This operator ensures type safety through compile-time checks.

- `dynamic_cast<T>`: Utilized for safe downcasting in class hierarchies, converting a base class pointer or reference to a derived class pointer or reference. It employs Run-Time Type Information (RTTI) to verify the cast's validity, returning a null pointer or throwing a `bad_cast` exception upon failure.

- `reinterpret_cast<T>`: Enables conversions between pointer types or between pointer and integer types, performing a bit-wise copy of the value. Its use is generally discouraged due to potential alignment and type punning issues, and should only be considered when absolutely necessary.

- `const_cast<T>`: The only cast capable of modifying the constness of an object, useful for invoking non-const methods on const objects or modifying objects initially declared as const. This operator can add or remove the `const`, `volatile`, or `__unaligned` qualifier from a pointer or reference.

Transitioning from the basics of types and casting, let's delve into practical examples to illustrate these concepts further.

Examples:

```
// different static type-casting variants
short a = 10;
float b;
b = a;                      // implicit (variant 0)
b = (float) a;              // explicit (variant 1)
b = float(a);               // explicit (variant 2)
b = static_cast<float>(a); // explicit (variant 3)

// dynamic type-casting for pointers/references to objects
Animal* animal = new Dog();
Dog* dog = dynamic_cast<Dog*>(animal);

// reinterpreting memory, here array interpreted as __m128 vector:
float data[] = {1.f, 2.f, 3.f, 4.f};
__m128 &vec = reinterpret_cast<__m128&>(data);

// modifying constness of an object to re-write const-objects:
int c = 10;
const int& d = c;           // create "write-protected" reference
const_cast<int&>(d) = 5;    // re-write "write-protected" reference
```

1.2.7 Variables, Literals, and Expressions

In C++, *variables* are identifiers providing a name to objects in memory, for instance, `int a;` defines a variable named `a` of type `int`. *Literals* represent constant values directly included in the code, like `5` in `a = 5;`. An *expression* involves operators and operands performing computations, such as `a++`, which increments `a`.

Each object in C++ is characterized not just by its data type and size, but also by two main properties:

– An address, indicating the memory location of the object.

– Content, the data value stored at the object's memory location.

```
int a = 42; // Creates an integer variable named 'a' with a value of 42
a = 1337;   // Modifies the content of 'a' to 1337
```

1.2.8 Initialization in C++

C++ supports various initialization methods. Traditional forms include:

- Direct Initialization: `int a(5);`
- Copy Initialization: `int a = 5;`

C++11 introduced *Brace Initialization*, offering advantages such as a uniform syntax across all types, prevention of narrowing conversions, and uniform initialization of aggregates (arrays, structs):

```
int a{4};      // No error
int b{7.9};    // Compiler error due to narrowing
```

Initialization of data structures:

```
std::vector<int> v{1, 2, 3};
int array[]{1, 2, 3};
YourClass obj{};
```

This initialization method enhances code safety and is thus recommended in modern C++ practices.

1.2.9 Pointers in C++

Your introduction effectively sets the stage for understanding memory addresses. Clarifying that every memory cell can be addressed and that variables are allocated to these addresses could be beneficial. It might be helpful to mention that the size and alignment of variables affect how they are placed in memory, which is managed by the compiler and the runtime environment.
Any variables in C++ can be accessed with its identifier (name). This is the easiest way when we don't need to care about the physical location of our data in the memory. However, there is a second way to access a variable, directly using its location in the memory. The computer's memory can be represented as a sequence of memory cells of one byte each. These cells are numbered in a consecutive way. Each cell has a unique number in the whole available memory. Every next cell has the number of the previous one plus one. This way we can claim that the cell number 55 definitely follows the cell number 54. As soon as we declare a variable, the amount it needs in memory is allocated in a specific location (memory address). The operating system performs this task automatically during runtime. This memory address locates a variable in the memory. This memory address can be obtained by adding an ampersand sign (`&`), the so-called *address operator*, in front of the identifier. So `&a` gets the address of the variable a. A variable which stores the memory address of another variable is called a pointer.

Pointers are a foundational aspect of C++, allowing variables to reference memory locations of objects. Key points regarding pointers include:

- Pointers have a consistent size, typically making them more efficient to move or copy compared to large data structures.

- Modifying an object via a pointer affects the original object globally.
- Pointers facilitate data structure manipulations, such as swapping nodes in trees without moving all child nodes.

Pointers are said to ”point to” the variable whose address they store. All pointers have a special pointer type. While declaring a pointer, one has to specify this type. The pointer type is obtained by adding an asterisk after the type it points to.

```
int a = 5;    // 5 stored at some memory location, let's say at 0x7ffcdf8a990c
int* b = &a;  // 'b' points to the address of 'a' (0x7ffcdf8a990c)
              // 'b' itself is located somewhere else, e.g. at 0x7ffcdf8a9900
```

The content of a pointer is a memory address and the address where the pointer is pointing to has some content, which can be accessed by dereferencing the pointer.

Dereferencing Pointers: The value (content) at the pointer's referenced memory location can be accessed by dereferencing the pointer. One can use the asterik sign (*) in front of the pointer variable to dereference the pointer. In this case, the * is called *dereference operator*.

```
std::cout « ”address of a: ” « &a « ”, content of b: ” « b « std::endl;
std::cout « ”content of a: ” « a « ”, dereferenced b: ” « *b « std::endl;
std::cout « ”address of b: ” « &b « std::endl;
```

Additionally, operations can be applied to pointers, like arithmetic operations to traverse arrays.

```
int arr[2];
int* p = arr; // same as ”int* p = &arr[0]” -> p points to the first element
++p;          // p points now to the second element
```

In this example, the behavior of `arr` is similar to pointer: one can either set `p` to `arr` or to the address of element 0 of the array, but not to the address of `arr`. It might seem like the content of `arr` is an address, similar to pointers, but actually `arr` is a array-type. But arrays in expressions decay to pointers to their first element, reflecting a design choice from C for simplicity and efficiency.
This conversion facilitates passing arrays to functions as pointers, enabling efficient data manipulation. However, a critical implication of array decay is the loss of size information. When an array decays to a pointer, the function receiving the pointer does not inherently know the size of the original array. This necessitates careful management, often requiring the size of the array to be passed as an additional parameter to functions that operate on the array. Understanding and accounting for array decay is essential to avoid out-of-bounds access, ensuring safe and effective use of arrays in C++. In modern C++, other container classes/structs are usually used, such as `std::vector<T>` or `std::array<T>`.

Fundamentally, a pointer is designed to hold the address of a single memory location. However, when it references an object occupying more than one byte in memory – such as a multi-byte

data type or an array – the pointer is actually pointing to the initial byte of a contiguous block of memory where the object resides. For instance, a pointer referencing a 4-byte integer, or an array thereof, essentially points to the first byte of this integer. When such a pointer is incremented, it advances to the next element in the sequence. The magnitude of this advancement, or ”jump”, is determined by the size of the object type to which the pointer refers. Consequently, for an array of 4-byte integers, incrementing the pointer results in it moving 4 bytes forward, aligning with the start of the subsequent element in the array. Thus, pointer-arithmetic has type-awareness.

Please note: The `this` keyword in C++, representing the object itself, is also a pointer.

1.2.10 References in C++

References in C++ create an alias for an existing variable, serving as a powerful tool in high-performance computing by circumventing the need for costly copies. They find their primary utility in two scenarios: as return types and as function parameters, significantly enhancing performance by operating directly on the original objects.

Consider the distinction between working with a copy versus a reference:

Working with a copy:

```
Matrix matrix = obj.GetLargeMatrix(); // Incurs significant overhead by copying
matrix.DoSomething();
```

Working with a reference:

```
Matrix& matrix = obj.GetLargeMatrix(); // Efficiently accesses the original
matrix.DoSomething();
```

To utilize a reference, one has to change the function’s return type to return a specific reference type, e.g. `Matrix& GetLargeMatrix()`.

This technique is similarly applied to function parameters to ensure that operations are performed on the original data without incurring the overhead of copying. Function parameters declared with an &-sign are treated as references, enabling direct manipulation of arguments as if they were the originals. This approach is not only efficient but also maintains code clarity and intuitive logic flow. However, it is crucial to ensure that returned references do not point to local objects within a function scope to avoid dangling references and the errors resulting from this.

```
Matrix MatrixMultiplication(const Matrix& a, const Matrix& b)
```

Here, two references of type `Matrix` are expected as function parameters. In a non-static function that is applied directly to an object, one matrix (e.g. Matrix ’a’) could be the object the function is applied to and then only one matrix (e.g. Matrix ’b’) would be given as a parameter.
However, this is somewhat a design choice. In some cases, in high-performance computing, you will also see that the result reference is given as a function parameter. For example, when memory is already pre-allocated for re-using it multiple times, such a concept makes sense and

avoids multiple re-allocations of large spaces in memory. In the end, this is usually a trade-off between intuitive code and performance. For maintenance reasons, you should not always opt for performance, but in limited environments, it might help.
The return type here is not a reference, as in this example, it is assumed that the result is a new matrix and not one of the two operands is overwritten. When creating objects inside a function body that is then not stored in another object with a longer lifetime, one can not return a reference to that object. When the function body (scope) ends, the lifetime of the new object ends and the reference would be referring to an object that does not exist anymore, leading to an error.
Additionally, in the example above, the `const` keyword is used to avoid accidental modifications to the matrices. The keyword will be discussed in more detail later on.

A reference is implemented similarly to a pointer (e.g. it has a small size of 8 bytes in 64-Bit systems), while a copy is always depending on the target object's size.

When working with large data structures, copying a large object is timely expensive and should be avoided whenever possible.

1. Memory has to be allocated for the new object
2. The whole object has to be copied into the new location

Avoiding unnecessary copies not only saves time but also reduces memory usage, which is critical in high-performance computing and memory-constrained environments like embedded systems.

1.2.11 Const Correctness

Ensuring `const` correctness in function parameters and return types is a best practice in C++ programming. This involves using `const` with references to prevent modification of the original object, enhancing code safety and expressiveness.

Example:

```
1 void DisplayMatrix(const Matrix& matrix) {
2     // Display the matrix without modifying it
3 }
```

Applying `const` to reference parameters enforces immutability, allowing read-only access to the object. This guarantees that the function will not alter the object's state, providing clarity to other developers and preventing unintended side effects.
Furthermore, `const` correctness aids in the optimization of compiler-generated code, potentially unlocking performance enhancements through the assurance that certain objects remain unchanged throughout their use. This can also have a non-negligible effect on the runtime.

Const References as Return Types

While returning a reference can significantly optimize performance by avoiding unnecessary copies, combining it with `const` ensures the returned object is protected against unintended modifications. This technique is particularly useful when returning objects that are members of a

class or have a static lifetime, ensuring the safety and integrity of the returned reference.

Example:

```
const Matrix& GetReadOnlyMatrix() {
    static Matrix matrix;
    // Perform operations
return matrix; // Returns a read-only reference
}
```

Incorporating `const` correctness into your C++ programming practices promotes not only efficiency and resource conservation but also contributes to the development of robust, error-resistant code.

1.2.12 Control Flow

C++ allows the programmer to have different paths of execution through the program. This is achieved via a collection of control flow statements. The simplest example is perhaps the if-statement that lets us execute a statement only if a conditional expression is true. For ease of understanding we will divide control flow statements into six categories: conditional statements, jumps, function calls, loops, halts and exceptions.

Conditional statements cause a block of code to be executed if and only if a certain condition, such as an equality (==) or inequality (>,<, <= etc.), is satisfied. The if-else and switch statements fall into this category. Jump statements tell the CPU to execute statements at some other location in the program. The break, continue and goto statements are jump statements. Loops tell the program to repeatedly execute some sequence until some condition is met. The while, do-while, for and ranged-for statements are used to generate loops. Exceptions are a special kind of statements designed for error handling such as unexpected user input. C++ uses try, throw and catch for exception handling. You will not encounter many uses of exceptions in this course, but the exception handling is important generally. Function calls are jumps to some other location in the code and back. Function calls and return are used together to achieve this effect. Finally, In some situations the only recourse is to terminate the program. Halt statements like std::exit() and std::abort() statements can be used for this.

C++ allows the programmer to nest control flow statements i.e. it is allowed to use, for example, loops inside loops which may themselves be inside of an if statement. This nesting of control flow statements allows the program to take highly convoluted and input dependent paths. A good command over control flow statements is absolutely necessary to follow the rest of this course. Please spend time to master these statements.

The following example provides at least some applications of conditional statements and loops.

```
int a = 5;
int b = 10;
int c = (a > b) ? b : a--;

```

```
for (int i = 0; i < c; i++) {
    do {
        std::cout « "a: " « a « std::endl;
        a--;
    }
    while (a > -1);

    std::cout « "i: " « i « std::endl;
}
```

1.2.13 Stack vs. Heap

There are primarily two types of memory allocations in C++: stack and heap allocations. For stack allocation, the allocation happens on contiguous blocks of memory. We call it a stack memory allocation because the allocation happens in the function call stack. The size of memory to be allocated is known to the compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is deallocated. A programmer does not have to worry about memory allocation and deallocation of stack variables. In the case of heap, the memory is allocated during the execution of instructions written by programmers. Every time when we make an object using the operators for dynamic allocation, it is always created in heap-space and the referencing information to these objects is always stored in stack-memory. Heap memory allocation is not as safe as Stack memory allocation because the data stored in this space is accessible or visible to all threads. If a programmer does not handle this memory well, a memory leak can happen in the program.

```
int a = 5;              // int allocated on stack
int* b = new int(3);    // int allocated on heap

// print out "5 3":
std::cout « a « " " « *b « std::endl;

delete b;               // free heap-allocated memory manually
```

1.2.14 Classes and Structs

Classes and structs are the constructs whereby the user can define their own types. The two constructs are identical in C++ except that in structs the default accessibility is public, whereas in classes the default is private. The term accessibility here refers to the allowed access to the members of the class/struct. Access controls enable you to separate the public interface of a class, which can be interacted with "directly" after an object from the class is created, from the private implementation details, which are set during object creation and changed using member functions of the class, and the protected members that are only for use by derived classes. The access specifier applies to all members declared after it until the next access specifier is encountered. Classes and structs can both contain data members and member functions, which enable you

to describe the type's state and behavior. Structs, however, are often used to create small data structures, whereas classes are often used for more complex objects.

```
struct Node {
    int id;
    int value;
};

struct Edge {
    Node* u;
    Node* v;
};


class Graph {
private:
    int nNodes_;
    int nEdges_;

    Node* nodes_;
    Edge* edges_;
protected:
    // member functions and variables
public:
    Graph(Node* nodes, Edge* edges, int nNodes, int nEdges) :
        nodes_(nodes),
        edges_(edges),
        nNodes_(nNodes),
        nEdges_(nEdges)
    {
        // other constructor code
    }

    // member functions and variables
};
```

When working with classes in multiple header and source files, it is important to set guards to ensure that every class is included once. This can be done by the following guards:

```
#ifndef CLASSNAME_H
#define CLASSNAME_H

class CLASSNAME
{
    // class code here
};
```

```
8
9  #endif // CLASSNAME_H
```

1.2.15 Templates

The template system is designed to simplify the process of creating functions (or classes) that are able to work with different data types. Instead of manually writing a bunch of almost identical functions or classes (one for each set of different types), we can instead create a single template. Just like a normal definition, a template describes what a function or class looks like. However unlike a normal definition (where all types must be specified), a template uses one or more placeholder types. A placeholder type represents some type that is not known at the time the template is written, but that will be provided later. Once a template is defined, the compiler can use the template to generate as many overloaded functions (or classes) as needed, each using different actual types! Thus we only have to create and maintain a single template, and the compiler does all the hard work by replacing the types respectively. In fact, the C++ standard library itself is absolutely full of template code.

To create a template function, replace instances of specific types (such as int or float) with type template parameters which we name, for instance, T. Now we have to declare the template parameter to let the compiler know that the function is a template. We do this by adding the following line before the function declaration: template <typename T>.

```
1  // function template
2  template <typename T>
3  T GetMax (T a, T b) {
4      if (a > b) return a;
5      return b;
6  }
7
8  // class template
9  template <class T>
10 class Pair {
11 private:
12     T a, b;
13
14 public:
15     Pair (T first, T second) : a(first), b(second)
16     {
17         // other constructor code
18     }
19
20     T GetMax()
21     {
22       if (a > b) return a;
23       return b;
```

```
24      }
25  }
```

1.2.16 Multi-File Projects and (Meta-)Build-Tools

Many projects consist of more than a single source code file. There are several tools that can be used to handle projects. A well-known tool is CMake, that allows to create a configuration file with all relevant information, such as files to compile, compiler settings, dependency checks and much more. In the following example, we assume a project in the `project` directory. In this directory, there is the CMake configuration `CMakeList.txt` that contains all relevant information required for compiling the application. Depending on the project structure, there are code files or other directories with the code included that are then loaded by the `CMakeList.txt`.
To compile our project, a `build` directory is created using the "make directory"-command `mkdir`. A second step is to navigate into this directory using "change directory"-command `cd`. Then we use the CMake tool to setup our code for compilation, using the `CMakeList.txt` in the directory outside of the build directory - CMake will automatically search for it. After this, we can run `make` to build our project and run our application.

```
username@hostname:~/project$ mkdir build
username@hostname:~/project$ cd build
username@hostname:~/project/build$ cmake ../.
username@hostname:~/project/build$ make
username@hostname:~/project/build$ ./PROJECTNAME
```

A simple `CMakeLists.txt` file could look like this:

```
cmake_minimum_required(VERSION 3.22)
project(PROJECTNAME)

set(CMAKE_CXX_STANDARD 14)

include_directories(.)

add_executable(PROJECTNAME
    Project.cpp
    ClassA.h
    ClassA.cpp
    ClassB.h
    ClassC.cpp)
```

Chapter 2

Neural Networks

2.1 Introduction

Neural networks, also known as Artificial Neural Networks (ANNs), are a subset of machine learning and are a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain. It is a type of machine learning process that uses interconnected nodes (neurons) in a layered structure, similar to neurons and synapses in a brain. It creates an adaptive system that computers use to learn from their mistakes and improve continuously. Thus, ANNs attempt to solve complicated problems, like summarizing documents or recognizing faces with high accuracy, and build the heart of deep learning algorithms.

Neural networks, along with other machine learning algorithms, have become popular across various scientific disciplines. A fundamental example of neural networks is the Multi-Layer Perceptron (MLP), which is often deemed a foundational neural network, since many advanced neural network models derive from it. This MLP has its roots in the simple perceptron, widely recognized as the cornerstone of neural network theory.

MLPs comprise of node layers, containing an input layer, one or more hidden layers, and an output layer. Each layer stores information or basically, a set of values, and each of the values is called a neuron. Each neuron is connected to at least one other neuron and has an associated weight and a threshold to be active or not. The output of each node is then activated before passing on to the next layer. A common type of activation used is to check if the output of any individual node is above the specified threshold value. In this case, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network. Modern neural networks, however, often apply mathematical functions to neurons or layers that modify the output values respectively to get a desired effect.

Neural networks rely on training data to learn and improve their accuracy over time. However, once these learning algorithms are fine-tuned for e.g. accuracy, they are powerful tools of computer science, allowing us to classify and cluster data at a high velocity. Tasks in speech recognition or image recognition can take minutes versus hours when compared to the manual identification by human experts. One of the most well-known neural networks is Google's search algorithm.

2.2 Multilayer Perceptrons

Among the various types of ANNs, in this chapter, the focus is on Multi-Layer Perceptrons (MLPs) with backpropagation based learning algorithms. These belong to the supervised learning, a subset of machine learning where the desired output for training is known exactly and that allows to provide accurate feedback for the network by the true labels (target values). It means that the network builds a model based on examples in data with known true outcomes. An MLP has to extract this relation solely from the presented examples, which together are assumed to implicitly contain the necessary information for this relation. The known true output is then used to provide feedback and adjust the weights accordingly.

As mentioned at the beginning an MLP comprises of three types of layers (input, hidden, and output) with nonlinear computational elements (neurons). The information flows from the input layer to the output layer through none, one or more hidden layer(s). In the common MLPs, the layers are often fully-connected layers. This means that all neurons from one layer are connected to all neurons in the adjacent layers as shown in Figure 2.1. These connections are represented as weights in the computational process. The weights play an important role in the propagation of the signal in the network. They contain the knowledge of the neural network, about the problem-solution relation. The number of neurons in the input layer depends on the number of independent variables in the model, whereas the number of neurons in the output layer is equal to the number of dependent variables. The number of output neurons can be scalar, vector or higher dimension.

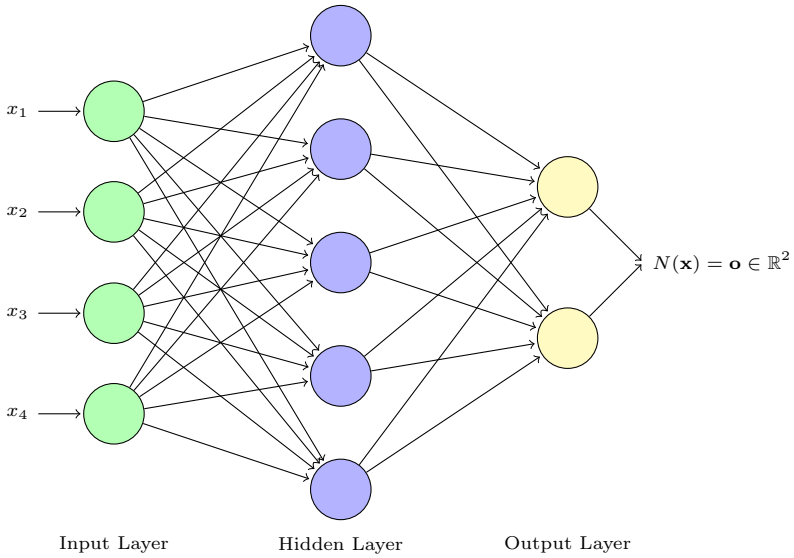

Figure 2.1: MLP with 4 input neurons, 5 hidden neurons and 2 output neurons.

2.3 Learning Algorithm

An MLP is trained/learned to minimize errors between the desired target values and the values computed from the model. If the network gives the wrong answer, or if the errors are greater than a given threshold, the weights are updated to minimize them. Thus, errors are reduced and, as a result, future responses of the network are likely to be correct. In the model we will be learning based on this chapter, the network will be updated, more specifically the weights

would be updated, for both the correct and incorrect results so as to reach a global minimum with respect to its prediction capabilities. In the learning procedure, datasets of input and desired target pattern pairs are presented sequentially to the network. The learning algorithm of an MLP involves a forward-propagation step followed by a backward-propagation (BackProp) step.

2.4 Forward Propagation

The forward-propagation phase begins with the presentation of the input variable(s) **X** (which can be a vector, matrix or any order tensor) to the input layer. As in biological neural systems, in which dendrites receive signals from neurons and send them to the cell body, an MLP receives information through input and output neurons and summarizes the information. MLP training is based on an iterative gradient algorithm to minimize error between the desired target and the computed model output. The neuron *j* of the hidden layer shown in the figure below is calculated from the previous layer, where the summation of each output of the previous layer (the input layer in the figure below) multiplied by weight w_{ij} (which is the weight associated with the *i*th component of the previous layer and *j*th component of the current layer). However, note that this is not the final output of the neuron h_i.

$$a_j = \sum_{i=N_{input}} x_i^{input} w_{ij}$$

where N_{input} is the number of features of the input layer (you can also call them features, units, nodes etc).

In general we also add a *bias* to our calculation.

$$a_j = \sum_{i=N_{input}} x_i^{input} w_{ij} + b_j$$

This is followed by the application of an activation function and the final output becomes the neuron j of the hidden layer and will be used in the construction of the next layer. The activation function is almost always a non-linear function. It plays an integral role in neural networks by introducing nonlinearity. This nonlinearity allows neural networks to develop complex representations and functions based on the inputs that would not be possible with a simple linear regression model. Some of the most common activation functions are: sigmoid, tanh, and ReLU. All hidden layers usually use the same activation function. However, the output layer will typically use a different activation function from the hidden layers. The choice depends on the goal or type of prediction made by the model. Assume that f_A is some activation function. Then,

$$h_j = f_A(a_j)$$

For all the N_{hidden} units in the hidden layer we can represent the above summation in a compact form as such:

$$\mathbf{a}_{hidden} = \mathbf{W}_{hidden} \cdot \mathbf{x}_{input} + \mathbf{b}_{hidden}$$

and the activation function which follows as :

$$\mathbf{h}_{hidden} = F_A(\mathbf{a}_{hidden})$$

Here, the activation is applied piece-wise if **a** has components.

In general we can say that the units/features(as mentioned before they are also called neurons, nodes etc) of *n*th layer, $\mathbf{h}_n$ can be written as:

$$\mathbf{a}_n = \mathbf{W}_n \cdot \mathbf{h}_{n-1} + \mathbf{b}_n$$
$$\mathbf{h}_n = F_A(\mathbf{a}_n)$$

2.5 Loss Function

In general, for any optimization problems, we have an objective function that we minimize or maximize in order to obtain our desired output. For the MLP that we will be learning, we will be minimising this objective function and as such we will be calling it the *loss* function. You can also see terms like *cost*, *error* in other sources of text. So these terms will be used interchangeably in the following.

One of the simplest loss function is to get the predictions from the network, subtract it from the actual number (the true value) we wanted to get and square it:

$$\mathbf{L}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y - \hat{y})^2$$

where *N* is the total number of training data. In practice, and in the exercises, we will be using Loss functions over a sample of training data, referred to as *batch size*. More details on this will be given in the section on *Gradient Descent*

2.6 Backpropagation

Backpropagation is the algorithm for fast calculation of the gradient with respect to the weights and bias, so that they can be later optimized based on a gradient-based optimization procedure. The main goal of the backpropagation algorithm is to calculate the gradients or partial derivatives of the loss function with respect to the weights and bias in each layer.

For example: $\frac{\partial C}{\partial w_{ij}}$, $\frac{\partial C}{\partial b_j}$ are the gradients with respect to the weight associated with neuron *j* mentioned in the *Section 2.4* and its bias.

Since the gradients are calculated based of the loss function, it's calculation starts from the final layer, i.e, the output layer. After that, these output layer gradients are used to calculate the gradients of the layer before the output layer. This is repeated until the layer after the input layer. Since the input layer is the data, it doesn't have any weights or bias for its construction. In order to calculate these gradients from the final output layer, where the cost function is calculated, to the lower layers, we use the chain rule to derive lower gradients from higher ones.

$$\frac{\partial C}{\partial x_i} = [\frac{\partial C}{\partial x_n}, \frac{\partial x_n}{\partial x_{n-1}}, ..\frac{\partial xi+1}{\partial x_i}]$$

So, to demonstrate the overall the backpropagation algorithm, consider an MLP with

$$(N + 2)$$

layers, which are more specifically, 1 input layer (the 0-th layer), N hidden layers and 1 output layer (the $(N+1)$-th layer). Let $L(X)$ be the *loss* function.

1. Compute the gradient on the output layer:
   $g = \nabla_y L$
   Calculate the gradient for the output layer and all lower levels.

2. **for** $n = N + 1$ to 1 **do**

   - Calculate gradient based on the layer features before activation. For this, just multiply the gradient of the layer with the derivative of the activation.
     $g = \nabla_{a_n} L = g \odot f'(a_n)$
     For eg:-, incase of the output layer in our example, this is just:
     $g = \nabla_{a_{output}} L = g \odot f'(a_{output})$
     , where the g on the RHS is gradient from the output layer($g = \nabla_y L$)

   - From this gradient we can calculate the gradient of the weights and bias used to create/construct this layer:
     $\nabla_{W_n} L = \nabla_{a_n} L \odot \nabla_{W_n} a_n = g \odot \nabla_{W_n} a_n$
     If we look back to **Section 1.2** on forward propagation, we can see that:

     $$\nabla_{W_n} a_n = g \odot \mathbf{h}_{n-1}$$

     $$\nabla_{b_n} a_n = g$$

     (Easily derivable from

     $$\mathbf{a}_n = \mathbf{W}_n \cdot \mathbf{h}_{n-1} + \mathbf{b}_n$$

)
   - Calculate gradients for the next layer:

     $$\nabla_{h_{n-1}} L = g \odot \nabla_{h_{n-1}} a_n$$

     $$\rightarrow \nabla_{h_{n-1}} L = W_n^T \cdot g$$

This summarizes the BackProp algorithm that we will be using for learning the MLP. For other type of neural networks, the main factor which changes is the formula for the gradient and change in the gradient formulation arising from the way the layer is constructed(for MLP it is simple matrix multiplication as seen in *Section 2.4*). At the core, we use chain rule and start from the final output layer and work our way backwards till all the gradients are calculated.

2.7 Gradient Descent

The neural networks that we will be learning are trained using the stochastic gradient descent algorithm. Stochastic gradient descent is an optimization algorithm that estimates the error gradient for the current state of the model using examples from the training dataset, then updates the weights of the model using the gradients calculated from backpropagation. So, we would have the form:

$$w_{ij} = w_{ij} - \eta \frac{\partial L}{\partial w_{ij}}$$

where w_{ij} again refers to the weight used in *Section 1.2*. This is a general formula and applies to all weights and bias and η is the learning rate.
The amount that the weights are updated during training is referred to as the step size or the "*learning rate*". Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0. The learning rate controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs given the smaller changes made to the weights each update, whereas larger learning rates result in rapid changes and require fewer training epochs. A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, whereas a learning rate that is too small can cause the process to get stuck.
In theory, the gradient calculation is done over the complete training data set. This is also known as *Batch Gradient Descent*. However, in this case, each epoch will only have one update. Although, this is guaranteed to converge to the Loss minima, we will have many epochs for this to happen. This can be very inefficient and slow because each iteration of the algorithm only improves our loss by some small step. There's also *Stochastic Gradient Descent* where we update after each example is passed through the neural network. Although, this method converges faster, it will more fluctuations in the cost function and the computations cannot be parallelized efficiently since we are using one example at a time.
In our exercise, and in most implementations, we use *mini-batch Gradient Descent* to solve this problem. It combines the capabilities of both *Batch Gradient Descent* and *Stochastic Gradient Descent*. The rule for updating the weights stays the same, but we're going to take on some small mini-batch of the dataset and use the mean gradient on that to update the weights. (**Note**: Some texts refer to *mini-batch Gradient Descent* as *Stochastic Gradient Descent* and use them interchangeably. We will do the same).

2.8 Important Key Words and (Hyper-)Parameters

Although machine learning isn't the primary focus of this lecture, it's beneficial to understand some key terms that frequently appear in this domain.

These terms include:

- Number of epochs, batch size, learning rate, and other hyperparameters
- Model descriptions
- Dataset splitting strategies for training, validation, and testing
- Underfitting and overfitting in model training
- Various parameter optimization techniques beyond standard gradient descent

- Regularization methods to improve generalization
- And more…

The following subsections provide definitions and explanations for each of these concepts.

2.8.1 Epochs

The term *epoch* refers to a single cycle through the entire training dataset. Since data is often limited, the model undergoes multiple epochs, adjusting weights iteratively to learn patterns within the data. Sometimes, a ”validation” or ”testing” phase is also conducted each epoch to evaluate performance on unseen data per epoch to see how the behavior on unseen data is, but a final test should be done with entirely new unseen data before model deployment.

Advanced optimization techniques like decaying learning rates benefit from shuffling data each epoch to prevent overfitting. Otherwise, the same data in the beginning of the dataset would always be learned with a higher learning rate then data in the middle or the end of the dataset.

2.8.2 Learning Rate

The *learning rate* determines the step size for each weight update. It multiplies the gradient value to control how much the model adjusts. Smaller learning rates allow for finer adjustments but may lead to longer training times and a risk of getting stuck in local minima. Larger learning rates may speed up training but risk overshooting optimal values, potentially never converging to a good solution.

2.8.3 Batch Size

The *batch size* is the number of samples processed before updating the model’s parameters. Different batch sizes lead to different gradient descent approaches:

- *Batch Gradient Descent:* Processes the entire dataset before updating weights, leading to high memory usage and slow progress.
- *Stochastic Gradient Descent:* Processes one sample at a time, resulting in faster updates but high variance in gradients.
- *Mini-Batch Gradient Descent:* Balances both by processing a small subset, allowing parallelization and smooth progress.

2.8.4 Defining Layers in Neural Networks

The rapid growth of machine learning research, particularly in neural networks, has led to inconsistencies in terminology due to the absence of universal conventions. A common example of this is the definition of a ”layer.” Some sources define a layer as a set of neurons without considering the connections between them, while others include weight matrices or even the outputs that feed into the next layer. This can lead to confusion, especially in cases where activation functions are not applied directly to input features. For example, if there’s no activation function on the input features, should we consider them an input layer? Maybe not, but they are multiplied by a weight matrix and generate outputs for the next layer. Assuming an identity activation function on input

features would mimic the behavior of typical MLP layers, so we might reasonably classify it as a layer.

In PyTorch, for instance, a linear layer is specified with `nn.Linear(in_features, out_features, ...)`. By this definition, we have a multi-layer perceptron with input features, a weight matrix, and output features. Combined with an activation function, we obtain a full neural network without hidden layers. Here, we might consider only one MLP layer, but if we view the input features as implemented neurons, we might interpret this setup as having two layers.

Ultimately, the interpretation depends on the context. In PyTorch, we might simply say we have one linear layer, not counting the input layer as a true "layer." Alternatively, in a more descriptive context, we might refer to these as input, hidden, and output layers.
For this lecture, we'll count each set of neurons as a distinct layer, whether it's designated as an input, hidden, or output layer. Regardless of approach, it's helpful to clarify definitions upfront to ensure consistent understanding in any specific context.

2.8.5 Dataset Splits

Datasets are commonly split into training, validation, and testing subsets:

- *Training Set:* Used to train the model, with the aim of capturing general patterns rather than memorizing specific examples.
- *Validation Set:* Used for tuning hyperparameters and evaluating different models during training, helping to select the best-performing version.
- *Test Set:* Reserved for final model evaluation to assess performance on unseen data, reflecting generalization ability.

2.8.6 Underfitting and Overfitting

Underfitting occurs when a model is too simple to capture the underlying patterns in the data, leading to poor performance on both training and test data. This typically occurs when the network is initialized with random weights but has not yet trained, or after training if the model lacks sufficient parameters to learn the underlying data patterns.

Overfitting happens when the model learns too many details of the training data, failing to generalize to new data. This often happens when using too many training iterations without applying optimization methods to prevent overfitting, or when the model has so many parameters that it memorizes the training data.

2.9 Optimization Techniques

2.9.1 Gradient Descent Variants

The basic gradient descent algorithm adjusts model weights based solely on gradient information. However, specific issues, such as getting stuck in local minima or overshooting minima, have led to several gradient descent variants:

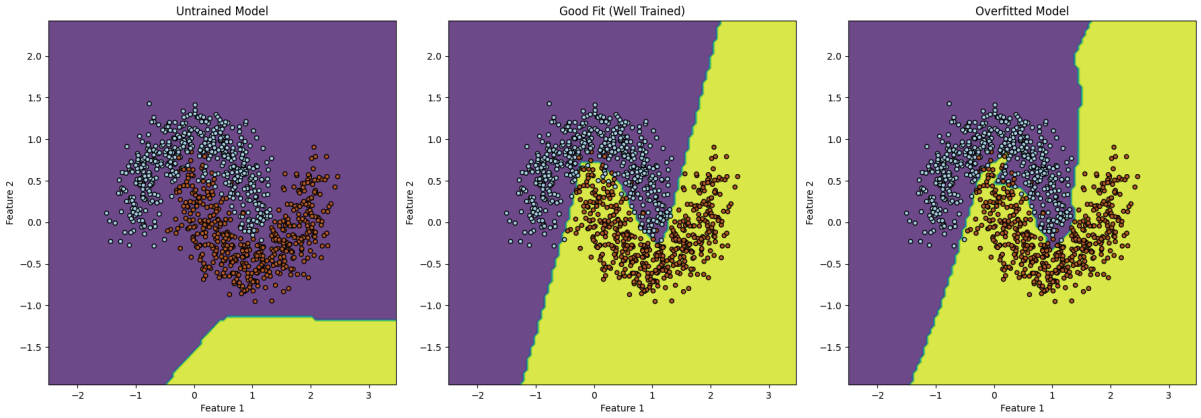

Figure 2.2: Classification example with an underfitted (untrained) model, a robust model and an overfitted model. The background color shows the decision space for one class or another. While the problem for the underfitted model is obvious, it is different for the overfitted one. There, the model tries to catch data points that reach into the cloud of the other class and thus, likely learning noise from that data. Adding more data will show for this example that these data points are ”outliers”. Trying to catch them, decreases the robustness for new unseen data.

- *Look-Ahead Methods:* Help avoid local minima by considering future steps.
- *Decaying Learning Rates:* Adjust learning rate over time to improve convergence.
- *Momentum-Based Updates:* Use past gradients to help navigate sparse features and handle zero-value paths.
- *Random Dropout:* Regularly drops random neurons during training, improving generalization and preventing overfitting.

2.9.2 Regularization Methods

Regularization techniques improve model generalization by discouraging complexity in the learned model:

- *L1 Regularization (Lasso):* Adds a penalty proportional to the sum of absolute weights. This encourages sparsity, leading to some weights becoming zero, which can simplify the model.
- *L2 Regularization (Ridge):* Adds a penalty proportional to the sum of squared weights. This discourages large weight values, preventing overfitting while retaining all features.
- *Dropout:* Randomly deactivates neurons during training, helping to prevent co-adaptation of features and enhancing generalization.

These methods can be adjusted based on the dataset and desired model behavior.

Chapter 3

Parallelization Methods

3.1 Moore's Law

Moore's Law is a prediction made by Gordon E. Moore, co-founder of Intel Corporation, in 1965. The law states that the number of transistors, resistors, diodes and capacitors on a microchip will double every 18 to 24 months, which leads to a rapid increase in computing power while decreasing the cost per component. Moore's Law has had a significant impact on the development of High Performance Computing (HPC) by enabling the creation of increasingly powerful and complex electronic devices that can handle large-scale scientific simulations and data-intensive computations. In the context of HPC, Moore's Law has been a driving force behind the development of faster and more powerful microprocessors, which have enabled researchers to perform simulations and calculations that were previously impossible. For example, the increase in computing power has allowed scientists to simulate complex physical phenomena like climate change, astrophysics, and molecular dynamics, which require massive amounts of computational power. However, recently, there are signs that Moore's Law may be reaching its limits. The physical limitations of semiconductor manufacturing technology, such as the size of the transistors, are becoming more apparent. As transistors become smaller and more densely packed, they generate more heat, which can lead to problems with reliability and performance. Furthermore, the cost of developing and manufacturing increasingly complex microchips is increasing, making it harder for companies to keep up with the pace of Moore's Law. As a result, some experts predict that the rate of growth predicted by Moore's Law may slow down or even come to a halt in the near future. This could have implications for the development of HPC, as it may become increasingly challenging to develop the more powerful processors required for large-scale simulations and computations. Despite these challenges, there are still efforts to keep Moore's Law alive, such as the development of new materials and manufacturing techniques. Additionally, there are emerging technologies such as quantum computing and neuromorphic computing that may enable the continuation of exponential growth in computing power beyond the limits of Moore's Law.

3.2 The Need for Parallelization

As the number of cores in a processor increases, the frequency of each core typically decreases to maintain thermal design power (TDP) limits. This presents a significant challenge for computational performance, as many applications are designed to run on a single thread with high frequency. One solution to this challenge is to develop applications that can take advantage

of multiple cores through parallelization. Parallelization involves breaking down a task into smaller sub-tasks that can be processed simultaneously on multiple cores. This can significantly increase performance by allowing the processor to perform more work in the same amount of time. However, parallelization is not always easy to implement, and not all applications are easily parallelizable. In some cases, parallelization is not possible due to dependencies or may require significant changes to the code or the use of specialized libraries or tools. Another solution to this challenge is the use of hardware accelerators such as Graphics Processing Units (GPUs). These devices are designed to perform specific types of computations much faster than a traditional CPU, making them ideal for computationally intensive tasks such as deep learning or scientific simulations. GPUs can also be used in conjunction with CPUs to offload certain types of computations and improve overall performance. For example, GPUs can be used to perform the computationally intensive parts of a simulation, while the CPU handles the rest of the computation. In general, parallelization or use hardware accelerators like GPUs are some solutions to the computational challenges posed by more cores with the same frequency.

3.3 Flynn's Taxonomy

Flynn's taxonomy is a categorization of forms of parallel computer architectures by Michael J. Flynn. Parallel computers are classified by the concurrency, or simultaneous calculations, in processing sequences (or streams), data, and instructions. This results in four classes:

- **Single Instruction Single Data (SISD)**
  A sequential computer which exploits no parallelism in either the instruction or data streams. Single control unit (CU) fetches single instruction stream (IS) from memory. The CU then generates appropriate control signals to direct single processing element (PE) to operate on single data stream (DS) i.e. one operation at a time. The traditional uniprocessor machines like a PC (currently manufactured PCs have multiple cores) or old mainframes.

- **Single Instruction Multiple Data (SIMD)**
  A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. For example, an array processor or graphics processing unit (GPU). This can be combined with other task-level parallization techniques to get further speedups.

- **Multiple Instructions Multiple Data (MISD)**
  Multiple instructions operate on one data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the Space Shuttle flight control computer.

- **Multiple Instructions Multiple Data (MIMD)**
  Multiple autonomous processors simultaneously executing different instructions on different data. MIMD architectures include multi-core superscalar processors, and distributed systems, using either one shared memory space or a distributed memory space.

3.4 Types of Parallelization

Parallelization is a technique that involves breaking down a task into smaller sub-tasks that can be processed simultaneously on multiple cores or threads. There are several types of parallelization

using threads and cores. However, to fully take advantage of these advanced programming models, we need to understand the basics of both paradigms. Threads and cores are both components of a computer's processing unit, but they serve different functions and have different capabilities.

Thread: A thread is a unit of execution within a process. Threads share the same memory space as their parent process and can execute instructions concurrently with other threads within the same process.

Core: A core is a physical processing unit that can execute instructions independently. It contains an arithmetic logic unit (ALU) for performing arithmetic and logical operations, as well as a control unit for managing the flow of instructions. A computer processor can have multiple cores, allowing it to perform multiple computations simultaneously.

Vector Unit: A vector unit is a specialized processing unit that is designed to perform mathematical operations on vectors or arrays of data in a single instruction. Vector units can perform a large number of operations in parallel, making them ideal for applications that involve large amounts of data such as scientific simulations, digital signal processing, and image processing.

The key differences between vector units, threads, and cores are their functions and capabilities. Cores are physical processing units that can execute instructions independently, while threads are units of execution within a process that can run concurrently with other threads on the same core or other cores. Vector units are specialized processing units that are designed to perform mathematical operations on vectors or arrays of data in a single instruction. Threads are also software constructs, while cores are physical hardware components. Threads rely on the underlying hardware, including the cores, to execute instructions. Multiple threads can be scheduled to run on a single core, allowing the core to perform multiple tasks simultaneously. However, each thread can only execute one instruction at a time.

3.5 SIMD Instructions in C++

As briefly mentioned in section 3.3, SIMD (Single Instruction Multiple Data) is a technique for performing data-level parallelism in a computer system. It involves the use of SIMD registers, which are specialized hardware units that can perform the same operation on multiple data elements simultaneously. SIMD is commonly used in applications that require large-scale data processing, such as multimedia, image processing, and scientific simulations. SIMD registers are designed to store multiple data elements of the same data type in a single register. For example, a 256-bit SIMD register can store 8 single-precision floating-point numbers of 4 bytes each or 4 double precision numbers with 8 bytes each. When an operation is performed on a SIMD register, the same operation is applied to all of the data elements stored in the register simultaneously. The use of SIMD registers allows for highly efficient data-level parallelism. Rather than performing the same operation on each data element individually, which can be time-consuming and inefficient, the operation can be performed on all of the data elements in the SIMD register simultaneously. This results in a significant increase in processing speed and efficiency. Data-level parallelism using SIMD registers can be achieved using a variety of programming languages and APIs, including SIMD extensions to C and C++, and libraries such as Intel Math Kernel Library (MKL) and OpenCV. These tools provide access to SIMD registers and the ability to write SIMD code that takes advantage of the parallel processing capabilities of the registers. For our course, we will be using Intel's Streaming SIMD Extensions (SSE)

intrinsic for the SIMD instructions. More specifically, we will be using a C++ class with all the common standard arithmetic operators overloaded for convenience. This class is implemented in the `P4_F32vec4.h` header file.

```
1    // creating a SIMD 0-vector
2    fvec vec = fvec(0.f);
3    // creating a SIMD vector with specific values
4    fvec vec = fvec(1.4f, 2.1f, 0.11f, 1.f);
5    // reinterpret cast dynamic float array to SIMD vector
6    float* arr = (float*) _mm_malloc(8 * sizeof(float), 16);
7    fvec* vec = reinterpret_cast<fvec*>(arr);
```

Listing 3.1: Examples for usage of `P4_F32vec4.h`.

In some cases, it can be challenging to rethink the implementation of a function when using SIMD. For example, the activation function LeakyReLU that is defined as

$$LeakyReLU(x) = \begin{cases} x & \text{if } x \geq 0, \\ 0.1x & \text{otherwise.} \end{cases} \tag{3.1}$$

Here, the function can easily be implemented by using a loop over all elements (neuron values) and a conditional statement if $x < 0$. Depending on the branch, the output will be set as defined. Using SIMD, however, it is more challenging. Assuming a 128-Bit vector unit, each neuron can keep track of 4 float values when SIMDized. Then, a conditional statement is not possible, since some vector elements could be negative, whereas others are positive or zero. One could use a loop to iterate over the SIMD vector elements, but that would be a sequential activation of elements. What can be done instead is using the tools provided by the SIMD extension libraries. There are several functions available, but an elementwise parallel condition statement is not easy to implement, especially due to branch prediction and pipelining. Instead, we use functions that are not depending on branches and conditions, even if it would be intuitive to make use of them. The concept of masks can sometimes be required in these situations. For example, a mask for all positive and negative values can be created. This can be done by using the sign-function of SIMD vectors, that returns a vector including elements that are either -1 or +1, depending on the sign of the provided input parameter vector's elements. This can be easily done by checking the sign bit of each element in parallel.

```
1    // get positive values incl. 0, all others are 0
2    fvec posMask = (sgn(values) + 1.f) / 2.f;
3    fvec posValues = (posMask * values);
4    // apply a factor of 0.01 to all negative values, all others are 0
5    fvec negMask = 1.0f - posValuesMask;
6    fvec negValues = (negMask * values) * 0.01f;
7    // set activated output values
8    values = posValues + negValues;
```

Listing 3.2: Example of rethinking from an SIMD perspective. Other solutions are possible as well.

Analyzing the code snippet, one could argue that the amount of calculations added due to the creation of masks reduces the speedup gained by SIMD, which is generally true. On the other hand, in this implementation no branch prediction (if $x < 0$) is required and the instructions can

be easily calculated without the risk of flushing the pipeline due to a mistakenly predicted branch outcome. Furthermore, considering a loop over all neurons, the values are arbitrary, making a branch prediction with high accuracy difficult. Thus, the runtime with SIMD can still be expected faster.

Chapter 4

Vector Class Library Vc

The Vector Class Library (Vc) is a C++ library designed to ease the use of SIMD (Single Instruction, Multiple Data) instructions. SIMD instructions allow parallel processing of data, significantly enhancing performance in computational tasks. However, directly using SIMD instructions such as __m128 can be cumbersome and error-prone, e.g. when moving to a new architecture where the vector register sizes differ. Vc addresses these issues by providing a high-level, easy-to-use interface while maintaining the performance benefits of SIMD.

4.1 Advantages of Vc over Hard-Coded SIMD Instructions

Vc offers several advantages compared to using hard-coded SIMD instructions:

- Abstraction: Vc abstracts the complexities of SIMD instructions, allowing developers to write clean and maintainable code.
- Portability: Code written with Vc is portable across different architectures and SIMD instruction sets (e.g., SSE, AVX). The library handles the underlying details.
- Performance: Vc optimizes the use of SIMD instructions, often providing performance improvements comparable to hand-tuned assembly code.
- Ease of Use: With Vc, developers can leverage familiar C++ idioms and template programming to write efficient SIMD code without needing deep knowledge of the hardware.

4.2 Vc in the C++26 Standard

The importance and utility of Vc have led to its anticipated inclusion in the C++26 standard. Some functionalities are already available in the standard library under `std::experimental`. However, not all features, such as scatter and gather operations, are currently available in this experimental version (https://github.com/VcDevel/std-simd).

```
#include <experimental/simd>
#include <iostream>

namespace stdx = std::experimental;

using float_v  = stdx::native_simd<float>;
```

```
int main() {
    alignas(16) std::array<float, 4> a_arr = {1.f, 2.f, 3.f, 4.f};
    alignas(16) std::array<float, 4> b_arr = {.1f, .2f, .3f, .4f};
    alignas(16) std::array<float, 4> c_arr;


    float_v& a = reinterpret_cast<float_v&>(a_arr[0]);
    float_v& b = reinterpret_cast<float_v&>(b_arr[0]);
    float_v& c = reinterpret_cast<float_v&>(c_arr[0]);

    c = a + b;

    for (size_t i = 0; i < c.size(); ++i) {
        std::cout « "c[" « i « "] = " « c[i] « std::endl;
    }

    return 0;
}
```

4.3 The VcDevel Library

For full functionality, including scatter and gather operations, the VcDevel library should be used. VcDevel is the foundation of Vc and provides all the advanced features that may not yet be available in the standard library (https://github.com/VcDevel/Vc

```
#include <iostream>
#include <Vc/Vc>

float input[N];
float output[N];

int main() {
    // TODO: fill input/output with values

    unsigned int index[float_v::Size];

    // example: gathering
    float_v values_vec;
    uint_v indices(index);
    values_vec.gather(input, indices)

    // example: masked gathering
    float_m mask = values_vec < 0.1f;
    float_v values_vec2(Vc::Zero);
    values_vec.gather(input, indices, mask);

    // example: masked scattering
```

```
    float_m mask2 = values_vec > .05f;
    values_vec.scatter(output, indices, mask2);

    return 0;
}
```

4.4 Conclusion

The Vc library offers a powerful and efficient way to utilize SIMD instructions in C++. Its abstraction and portability make it an excellent choice for high-performance computing tasks. While some features are available in std::experimental, the full capabilities of Vc can be accessed through the VcDevel library. As Vc becomes more integrated into the C++ standard, it is poised to become a fundamental tool for developers seeking to harness the power of SIMD in their applications.

Chapter 5

OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that enables parallel programming in shared-memory systems. It provides a simple and portable way to parallelize code written in C++, C, and Fortran. OpenMP allows developers to exploit the potential of multicore processors, maximizing performance by distributing workload across multiple processing units.
Parallelizing code can significantly improve the performance of computationally intensive applications. OpenMP simplifies the process of writing parallel programs by providing a set of directives, library routines, and environment variables. It allows developers to focus on identifying parallelizable sections of code and relies on the compiler and runtime system to handle the details of thread management and synchronization.
OpenMP offers several advantages for parallel programming:

- Simplified parallelization: OpenMP provides a set of directives, library routines, and environment variables that make parallel programming more accessible.
- Portability: OpenMP is supported by major compilers, making it easy to write parallel code that can be compiled and executed on different platforms.
- Incremental parallelization: OpenMP allows developers to gradually parallelize code by adding directives selectively, focusing on critical sections.
- Flexibility: OpenMP supports various parallelization models, including loop parallelism, task parallelism, and sections.

5.0.1 Compiler Support

To use OpenMP in C++, you need a compiler that supports OpenMP directives and library routines. Most modern compilers, such as GCC, Clang, and Visual C++, include OpenMP support. Once you have a compatible compiler, you can enable OpenMP by adding appropriate compiler flags, such as ”-fopenmp” for GCC and Clang or ”/openmp” for Visual C++. To use OpenMP in your code, you need to include the ”omp.h” header file. OpenMP supports several directives that are used to parallelize code, such as ”#pragma omp parallel” and ”#pragma omp for” which will be detailed below. These directives are used to define parallel regions and distribute loop iterations among multiple threads.

5.0.2 OpenMP Directives

OpenMP directives are pragmas that allow you to specify parallelism in your code. They guide the compiler in generating parallel code and define how the work should be divided among threads. Some commonly used directives include:

- `#pragma omp parallel`: Specifies that the code block following it should be executed in parallel by a team of threads.
- `#pragma omp for`: Distributes the iterations of a loop among the available threads, executing them concurrently.
- `#pragma omp critical`: Marks a section of code as critical, ensuring that only one thread executes it at a time.
- `#pragma omp barrier`: Inserts a synchronization point, forcing all threads to reach that point before continuing execution.
- `#pragma omp sections`: Divides the code block into sections, each to be executed by a different thread.

A parallel region is a block of code that can be executed by multiple threads in parallel. To create a parallel region, you can use the `#pragma omp parallel` directive. This directive is followed by an open brace '' to start the parallel region and a closing brace '' to end it. All the code within the parallel region will be executed by multiple threads. A simple usage of `#pragma omp parallel` is illustrated below:

```
#include <iostream>
#include <omp.h>

int main() {
        int num_threads;

        #pragma omp parallel
        {
                #pragma omp single
                {
                        num_threads = omp_get_num_threads();
                }
        }

        std::cout « "Number of threads: " « num_threads « std::endl;

        return 0;
}
```

In the above example, the code within the parallel region will be executed by multiple threads. The "omp_get_thread_num()" function returns the ID of the current thread, allowing you to identify which thread is executing the code.
Another useful directive is the `#pragma omp for` directive to parallelize loops. By adding this directive before a loop, you can distribute the iterations of the loop among multiple threads.

OpenMP automatically divides the loop iterations and assigns them to different threads for parallel execution.

```
#include <iostream>
#include <omp.h>

int main() {
        const int N = 10;
        int sum = 0;

        #pragma omp parallel for
        for (int i = 0; i < N; ++i) {
                #pragma omp atomic
                sum += i;
        }

        std::cout « "Sum: " « sum « std::endl;
        return 0;
}
```

In the above example, the loop iterations are divided among the available threads, and each thread computes a part of the sum. The `#pragma omp atomic` directive ensures that the updates to the shared variable sum are performed atomically to avoid race conditions.

5.0.3 OpenMP Library Routines

OpenMP provides a set of library routines that allow you to control and query aspects of parallel execution. These routines can be used to manage threads, obtain information about the execution environment, and control thread synchronization. Some commonly used library routines include:

- `omp_get_num_threads()`: Returns the number of threads in the current parallel region.
- `omp_get_thread_num()`: Returns the thread number of the calling thread.
- `omp_set_num_threads()`: Sets the number of threads to be used in subsequent parallel regions.
- `omp_get_wtime()`: Returns the wall-clock time in seconds.

5.1 Data Sharing and Synchronization

When parallelizing code with OpenMP, it is crucial to consider data sharing and synchronization. By default, variables defined outside a parallel region are shared among all threads, which can lead to race conditions and data inconsistencies. OpenMP provides mechanisms to control data sharing, such as private variables, shared variables, and reduction clauses. Additionally, synchronization constructs like barriers and critical sections help coordinate thread execution and prevent race conditions.

5.2 Performance Considerations

While OpenMP simplifies parallel programming, it's essential to consider performance implications. Load balancing, data dependencies, and overhead due to thread synchronization can affect the scalability and efficiency of parallel code. Profiling tools and performance analysis can help identify bottlenecks and optimize parallel implementations.

5.3 Summary

This chapter introduced OpenMP, a powerful API for parallel programming in shared-memory systems. We discussed its benefits, directives, library routines, data sharing, synchronization, and performance considerations. OpenMP simplifies the process of parallelization by providing a portable and straightforward approach to exploit multicore processors.