CAB402 Programming Paradigms
Assignment 1

Due: 11th April 2025

Worth: 30%

The purpose of this assignment is to develop your functional programming skills by developing a non-trivial application using F#. The principal focus will be on learning to program in a "*pure*" functional style – i.e., without making use of any mutable state. In part 2 you will also contrast this pure functional style of programming with the more traditional *imperative* (impure) and *object-oriented* paradigms.

About Large Language Models

Large Language Models (LLMs) are a type of artificial intelligence designed to understand and generate human-like text. *Generative AI* refers to AI systems that can create new content, such as text, images, music, or even code. These systems learn patterns from existing data and use this knowledge to generate new, similar content. The key technology behind LLMs is the *Transformer architecture*, which uses self-attention mechanisms to process and generate text efficiently.

State-of-the-art commercial Generative AI models include ChatGPT from OpenAI. *LLaMA (Large Language Model Meta AI)* is an example of an open-source LLM developed by Meta. It aims to provide a more accessible and efficient alternative to proprietary models. *LLaMA 2* was released in July 2023 as the successor to LLaMA. LLaMA 2 offers models with 7B, 13B, and 70B parameters. These models and many others can be found on the Hugging Face website.

llama.cpp is an open-source implementation of the LLaMA 2 architecture, aiming to achieve state-of-the-art performance on a wide range of hardware, including GPUs. You can find more about it on GitHub.

Andrej Karpathy created karpathy/llama2.c, a simpler implementation of the LLaMA 2 architecture, with the goal of creating something simple and easy to understand rather than focusing on performance.

Wayne Kelly has then ported Karpathy's C implementation to C# and made further simplifications to aid understanding. You have been provided with that C# implementation of the LLaMA 2 architecture. The model that we will be using is the 15M parameter model from karpathy/llama2.c which was trained on the TinyStories dataset.

What do you need to do?

Part A: Pure F# (24 marks):

Create a pure functional F# implementation of the provided CSharpTransformer C# project, based on the F# skeleton solution provided. Ensure that you do not alter any function signatures and that your implementation passes all unit tests and test examples. For this part, you need to create a pure functional program that does not mutate any data structures. Aim to program in the style presented in the lectures, which means preferring higher-order functions over recursion and using the F# pipe operator to chain together sequences of higher-order functions involving simple lambda functions. Above all, your goal is to make your code elegant, simple, and easy to follow and maintain.

Part B: Impure F# (2 marks):

Create a copy of your project from Part A and explore how making use of impure mutable features of F# might increase the performance of specific functions that are known to take a relatively long time to execute. The provided C# implementation may provide some inspiration here in terms of what might be done more efficiently if mutation is allowed. You don't need to change the entire program, just provide an example of adding mutation for the purposes of efficiency. You do not need to maintain any of the structure or function signatures from Part A, so it is not expected that the unit tests created for Part A will be applicable here.

What you need to Submit

You should submit a Visual Studio solution that includes all of the original projects provided in the Skeleton solution, plus an additional project created for Part B. It is recommended that you create this new project for Part B by making a copy of the folder containing your Part A project, renaming the new folder and Visual Studio Project file (.fsproj) and add this new project to your Visual studio solution (.soln). To reduce file size when submitting you should remove the following:

- model.bin
- tokenizer.bin
- .vs folder
- bin folders
- obj folders

Everything remaining should be added to a compressed zip file named **SourceCode**.zip. Note: this must be a zip file, not a tar file, not a rar file, not an iso file, etc

In addition to submitting this source code, you should also submit a PDF document containing a **Written report** of 1 to 2 pages (worth 4 marks) that discusses your experience and compares the three approaches (F# pure functional, F# impure functional and C# object-oriented imperative). Your comparison should include a discussion on the efficiency and effectiveness of the three approaches. You should discuss which was easier to program, which code is easier to read, understand and maintain, which is more concise, and which produces more efficient code. Provide measures were applicable to support your claims.

See separate rubric document for assessment criteria!

Tips:

The following is a complete list of all functions that I used in my sample solution. You don't need to use the exact same set of functions, but you might find some useful functions in this list which you didn't know about previously.

- `Array.init`
- `Array.fold2`
- `Array.length`
- `Array.max`
- `Array.map`
- `Array.map2`
- `Array.mapi`
- `Array.maxBy`
- `Array.sum`
- `List.fold`
- `Seq.iter`
- `Seq.unfold`
- `System.Math.Exp`
- `System.Math.Sqrt`

Suggested order of implementation

Several of the functions in your F# implementation will depend on one another, and it will not be possible for you to pass unit tests for these functions without first implementing the functions that they call. You can complete the implementations in any order you like, the list below is only a suggestion to avoid ending up the situation where you cannot test the function you just wrote.

HelperFunctions

1. `add, elementWiseMultiply, matrixMultiply`
2. `rootMeanSquareNormalize, softMax, sigmoidActivation`
3. `rotateOneHead`
4. `rotateVector`

Attention

1. `attentionScore`
2. `weightedAttention`
3. `attentionForOneHead`
4. `attention`

Transformer

1. `createLookupFunction`
2. `feedForwardOneLayer`
3. `feedForwardAllLayers`
4. `feedForward`
5. `generateNextToken`