TECHNICAL UNIVERSITY OF ATHENS
DEPARTMENT OF MECHANICAL ENGINEERING
MSC IN AUTONOMOUS CONTROL SYSTEMS AND ROBOTICS

ADVANCED MANUFACTURING SYSTEMS
(CIM - INDUSTRY 4.0)

DESIGN AND IMPLEMENTATION OF AN LSTM-BASED NETWORK FOR COMPUTER AIDED PROCESS PLANNING SYSTEM

Ronaldo Tsela - 02124203
Fall 2024-2025

Contents

List of Figures

List of Tables

1. Purpose

The purpose of this project is to design, develop, train and validate a neural network - based computer aided process planning system (CAPP) for automated manufacturing process generation. This work forms part of the last project in the "*Advanced Manufacturing Systems - (CIM - Industry 4.0)*" lecture series for the MSc program in "*Autonomous Control Systems and Robotics*" at NTUA.

2. Overview

This document describes the design, usage, and evaluation of the CAPP system based on Long-Short Term Memory (LSTM) cells for predicting equivalent process chains for manufacturing a part. The implementation consists of a Python package designed to provide an easy to use API for setting the system up, training and evaluating the model and also generating up to four equivalent process chains for a given part. The system is evaluated using. This document is accompanied with also the software sources, where additionally to the package itself, example programs are included as templates for interested users to explore and test the system.

3. Introduction

In modern manufacturing, products are characterized by increased complexity, shorter lead times, and a growing demand for customization. Consequently, the need for efficient, accurate, and cost-effective process planning methods has become increasingly apparent. Computer-Aided Process Planning (CAPP) serves as a bridge between the design and manufacturing stages, assisting in the fabrication of components. CAPP systems can be categorized into two main types: variant and generative. Traditionally, process planning has relied heavily on expert knowledge and manual intervention, often leading to inconsistencies, delays, and human errors. However, modern generative CAPP systems, which automatically generate new process plans based on rules and algorithms, are becoming more prominent. Recent advancements show that data-driven generative CAPP approaches, based on machine learning technologies, can significantly improve the efficiency and decrease the cost of manufacturing parts.

This project presents the development and implementation of a CAPP system using Long Short-Term Memory (LSTM) cells to predict up to four equivalent process chains for manufacturing a specific part configuration. LSTM is a specialized form of Recurrent Neural Networks (RNNs) designed to learn and predict sequences with temporal dependencies. LSTM cells, with their ability to retain long-term dependencies while mitigating the issue of vanishing gradients, are particularly effective for sequence prediction tasks such as process planning.

This report describes the data used for modeling the problem and describes the methods employed to transform this data into input and training features for the neural network-based CAPP system. The application is implemented as a Python package with hardware and OS-agnostic properties, ensuring versatility and ease of use across different platforms.

4. Methodology

4.1. Data Preparation and Modeling

Like any other machine learning problem, this task requires data to build the model architecture. For this purpose, two datasets are provided containing part configurations and process plans for constructing these parts. Specifically each part is characterized by specific features, as outlined in the **Table 1** below. Additionally, each part in the dataset is associated with up to four equivalent process chains as mentioned. The available process chains are presented in **Table 2**.

It is important to note that each process can be used only once within a single process chain. This assumption is supported as well by the structure of the dataset, which consists of 1,176 entries, where each process listed in **Table 2** is executed only once. Furthermore, secondary processes are included to reinforce this constraint.

Table 1 - The part configuration encoding table

Part Feature Name	Linguistic Value	Numerical Value
Geometry	Pure axisymmetric Axisymmetric with prismatic features Prismatic Prismatic with axisymmetric features Prismatic with freeform features Freeform Unconventional	0 1 2 3 4 5 6
Holes	None Normal Normal threaded Normal functional Large Large threaded Large functional	0 1 2 3 4 5 6
External Threads	yes no	0 1
Surface Finish	Rough Normal Good Very good	0 1 2 3
Tolerance	Rough Normal Medium Tight	0 1 2 3
Batch Size	Prototype Small Medium Large Mass	0 1 2 3 4

Table 2 - Manufacturing processes list and encoding

Process Name	Numerical Value
Turning	1
Milling	2
5-Axis milling	3
SLM	4
Sand casting	5
High pressure die casting	6
Investment Casting	7
Turning Secondary	8
Milling Secondary	9
Hole Milling	10
5-axis Milling Secondary	11
Thread Milling	12
Tapping	13
Grinding	14
5-axis Grinding	15
Superfinishing	16
Drilling	17
Boring	18
Reaming	19
Special Finishing	20

In this project, the core of the system consists of the design and training of a neural network architecture to predict these manufacturing process chains given as input a part configuration description. Since computers cannot interpret linguistic values as humans do, the data must first be encoded into numerical values. There are several approaches for encoding, with the simplest being one-hot integer encoding. The encoded values for part configurations and processes are provided in **Tables 1** and **2** as well.

It is worth noting that while numerical labels for part configurations start from 0, process labels begin at 1. This is done intentionally to represent the absence of a process (process termination or no operation) using the 0 value.

However, neural networks are not particularly efficient at processing raw integer values with absolute values greater than 1 as they appear in these tables. Therefore, a scaling process is necessary to normalize these numbers to fall within the range [−1,1]. A standard scaling method is employed, where for each sample *x* the scaled value *z* is computed using the following formula:

$$z = \frac{x - \mu}{\sigma}$$

where μ is the mean value and σ and standard deviation of the training samples respectively. Once the input data is scaled, it is fed into the model (which can be viewed as a "black box" for now), activating its internal neurons and producing an output–forward pass–. For the input layer, six neurons are used to receive the encoded features. Proper weights are applied as computed from training.

4.2. The Architecture

The Long Short-Term Memory (LSTM) neural network is selected for its ability to effectively model time-series and sequential data. The network architecture includes a layer of LSTM cells with a time depth of 20 steps. To account for non-linear transformations, a fully connected layer, activated by a ReLU function, is added between the input layer and the LSTM layer creating in that way a simple deep network. The final output is produced through a fully connected layer with Softmax activation resulting in 21 output vectors. From each of the 20 vectors only the maximum value of the 21 outputs is retained, resulting finally in a 20 elements vector per head. A visual depiction of the architecture is provided in **Figure 1** as follows.

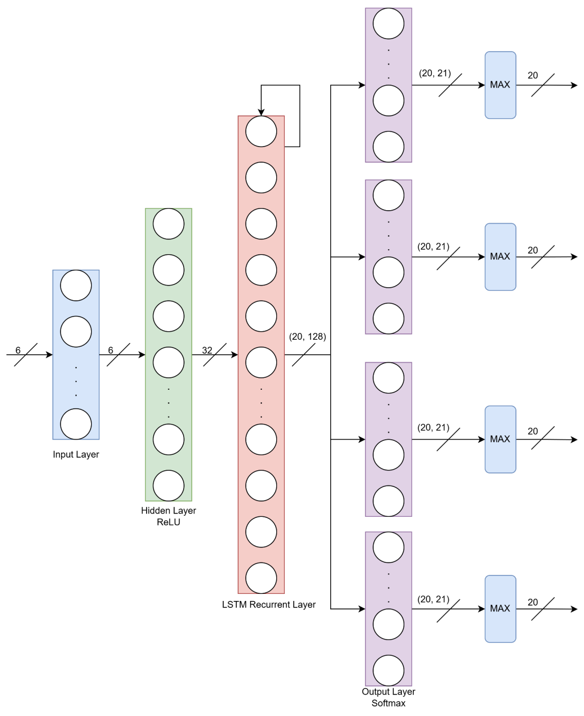
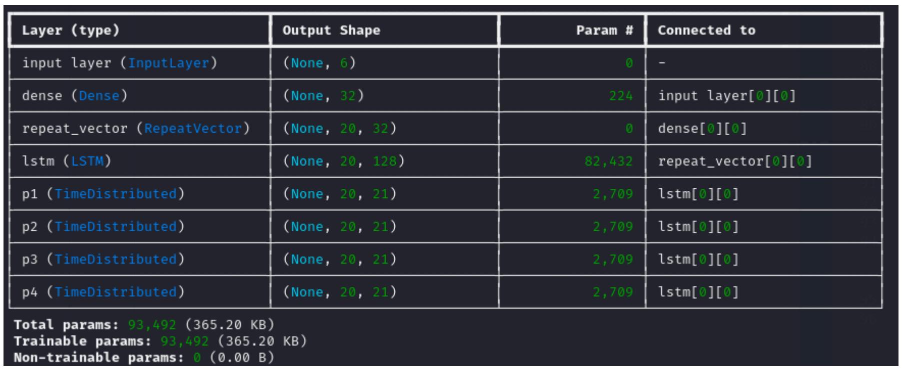


Figure 1 - The architecture of the proposed LSTM-based ANN.

4.3. Implementation

To develop this application, Python was chosen as the programming language due to its versatility, ease of use, and for its object-oriented nature coupled with TensorFlow's Keras framework for constructing, training and evaluating the artificial neural network architecture. Keras provides a developer-friendly API for designing and managing ANNs. Therefore the aforementioned architecture is implemented here in Keras with structure as described in the following **Figure 2**.



```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                  ┃ Output Shape           ┃      Param # ┃ Connected to            ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━┩
│ input layer (InputLayer)      │ (None, 6)              │            0 │ -                       │
├───────────────────────────────┼────────────────────────┼──────────────┼─────────────────────────┤
│ dense (Dense)                 │ (None, 32)             │          224 │ input layer[0][0]       │
├───────────────────────────────┼────────────────────────┼──────────────┼─────────────────────────┤
│ repeat_vector (RepeatVector)  │ (None, 20, 32)         │            0 │ dense[0][0]             │
├───────────────────────────────┼────────────────────────┼──────────────┼─────────────────────────┤
│ lstm (LSTM)                   │ (None, 20, 128)        │       82,432 │ repeat_vector[0][0]     │
├───────────────────────────────┼────────────────────────┼──────────────┼─────────────────────────┤
│ p1 (TimeDistributed)          │ (None, 20, 21)         │        2,709 │ lstm[0][0]              │
├───────────────────────────────┼────────────────────────┼──────────────┼─────────────────────────┤
│ p2 (TimeDistributed)          │ (None, 20, 21)         │        2,709 │ lstm[0][0]              │
├───────────────────────────────┼────────────────────────┼──────────────┼─────────────────────────┤
│ p3 (TimeDistributed)          │ (None, 20, 21)         │        2,709 │ lstm[0][0]              │
├───────────────────────────────┼────────────────────────┼──────────────┼─────────────────────────┤
│ p4 (TimeDistributed)          │ (None, 20, 21)         │        2,709 │ lstm[0][0]              │
└───────────────────────────────┴────────────────────────┴──────────────┴─────────────────────────┘
 Total params: 93,492 (365.20 KB)
 Trainable params: 93,492 (365.20 KB)
 Non-trainable params: 0 (0.00 B)
```

Figure 2 - The CAPP ANN architecture parameter table for 128 LSTM cells.

To train the model, the Adam optimizer with a constant learning rate set to 0.001 was utilized. The weights of each layer are updated using sparse categorical cross-entropy as the loss function, with the objective of maximizing the prediction accuracy for each of the four output heads. The training process is run for a specified number of epochs and evaluated using a separate dataset provided specifically for evaluation purposes. The evaluation data is prepared following the same methodology as the training data (as aforementioned). Each head is evaluated separately and results are depicted in **Figure 3**, in the **Results** section.

The entire application is provided as a Python package with the underlying implementation capable of training, evaluating and inferring the model with the user's needs.

5. User Manual

5.1. Package Installation

To keep this package separate from other Python packages on your system, it is best practice to use a virtual environment. To create and activate a virtual environment, on a terminal emulator run within the project folder the following:

```
$ python3 -m venv capp_lstm
$ source ./capp_lstm/bin/activate
```

Navigate to the *./capp_lstm_package* directory and run the following command to install the package locally on your computer (within the virtual environment you just created).

```
$ pip3 install -e .
```

Upon successful installation of the package and the required internal packages needed for the implementation, the package is ready for use. Make sure you have installed Python version 3.7 or greater in order for TensorFlow to operate properly!

5.2. Attributes and Methods

The entire system is implemented as a class named *CappLSTM*, which contains member functions (methods) and attributes that can be used to create, train, evaluate, and infer a specific model. The user-related attributes and methods are listed in **Table 3** below.

Table 3 - The methods and attributes of CappLSTM class.

Attribute/ Method	**Description**
attr: *training_dataset_path*	Specifies the path to the JSON dataset file used for training.
attr: *validation_dataset_path*	Specifies the path to the JSON dataset file used for validation during training.
attr: *evaluation_dataset_path*	Specifies the path to the JSON dataset file used for evaluating the model, separate from the validation set.
attr: *input_file_path*	Specifies the path to the JSON file containing a specific part description as input.
attr: *scaler_path*	Specifies the path to the exported scaler object (StandardScaler from Sklearn, saved in .ipk format).
attr: *model_path*	Specifies the path to the exported Keras model (in .keras format).
attr: *c_num_LSTM_cells*	Defines the number of LSTM cells used in the architecture. The default value is 16. This value can be increased for improved performance.
attr: *c_hidden_layer_size*	Defines the number of neurons in the hidden layer. The default value is 32. Increasing this value requires adding a dropout layer to prevent overfitting, which involves modifying the internal source code—not recommended.
attr: *batch_size*	Defines the batch size used for training when working with large datasets. The default value is 1 sample per batch.
attr: *training_epochs*	Specifies the number of training epochs to run.
attr: *eval_results*	Stores the evaluation results as a list of the form [[loss], [accuracy]] for each output head separately.
meth: *load_model*()	Loads a saved model from *model_path* and the corresponding scaler from *scaler_path*.

meth: *create_model*()	Creates a new model and saves it to *model_path* along with the corresponding scaler to *scaler_path*.
meth: *train*()	Trains the model using the training and validation datasets provided through *training_dataset_path* and *validation_dataset_path*, respectively.
meth: *evaluate*()	Evaluates the model independently from the validation phase performed during training.
meth: *training_results*()	Displays the learning curves and performance metrics from the training process.
meth: *predict*()	Infers the process chain for a specified part, using a JSON input file defined by *input_file_path*. The predicted process chains are returned in a human-readable format.

The following code snippets provide examples on how to build a new model, train a model, evaluate it, and infer it with a specific input.

5.2.1. New Model Creation Example

```
from capp_lstm_package.capp_lstm_package.capp_lstm import *

if __name__ == "__main__":
    capp = CappLSTM()

    capp.model_path = "/home/ronaldo/Desktop/CIM/capp-lstm/metadata/model.keras"
    capp.scaler_path = "/home/ronaldo/Desktop/CIM/capp-lstm/metadata/scaler.ipk"

    capp.c_num_LSTM_cells = 64

    capp.create_model()

    print(capp.model.summary())
```

5.2.2. Model Training Example

```
from capp_lstm_package.capp_lstm_package.capp_lstm import *

if __name__ == "__main__":
    capp = CappLSTM()

    capp.training_dataset_path = "/home/ronaldo/Desktop/CIM/capp-lstm/data/training.json"
    capp.validation_dataset_path = "/home/ronaldo/Desktop/CIM/capp-lstm/data/validation.json"

    capp.model_path = "/home/ronaldo/Desktop/CIM/capp-lstm/metadata/model.keras"
    capp.scaler_path = "/home/ronaldo/Desktop/CIM/capp-lstm/metadata/scaler.ipk"

    # Create a model first or load an existing one

    capp.train()

    capp.training_results()
```

5.2.3. Model Evaluation Example

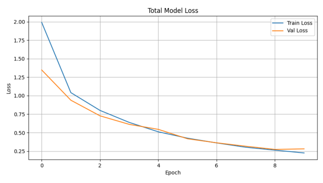
```
from capp_lstm_package.capp_lstm_package.capp_lstm import *

if __name__ == "__main__":
    capp = CappLSTM()

    capp.evaluation_dataset_path = "/home/ronaldo/Desktop/CIM/capp-lstm/data/validation.json"

    capp.model_path = "/home/ronaldo/Desktop/CIM/capp-lstm/metadata/model.keras"
    capp.scaler_path = "/home/ronaldo/Desktop/CIM/capp-lstm/metadata/scaler.ipk"
    capp.load_model()

    capp.evaluate()

    loss = []
    acc  = []

    total_loss = capp.eval_results[0]

    for i in range(1, 5):
        loss.append(capp.eval_results[i])
        acc.append(capp.eval_results[i+4])

    print(f"Total Loss: {total_loss}")
    print(f"Loss Vector : {loss} ")
    print(f"Accuracy Vector : {acc}")
```

5.2.4. Model Inference Example

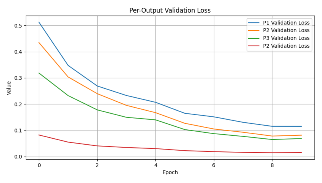
```
from capp_lstm_package.capp_lstm_package.capp_lstm import *

if __name__ == "__main__":
    capp = CappLSTM()

    capp.input_file_path = "/home/ronaldo/Desktop/CIM/capp-lstm/data/part_test.json"

    capp.model_path = "/home/ronaldo/Desktop/CIM/capp-lstm/metadata/model.keras"
    capp.scaler_path = "/home/ronaldo/Desktop/CIM/capp-lstm/metadata/scaler.ipk"

    capp.load_model()

    results = capp.predict()

    for i in range(0, 4):
        print(f"Process Chain {i} : {'->'.join(results[i])} ")
```

6. Experimentation and Results

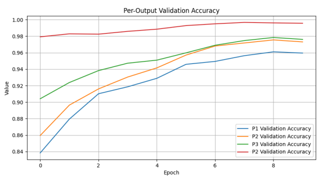
After creating a system with 128 LSTM cells and training it using the training set for 10 epochs, the learning curves are shown in **Figure 3**. The average accuracy achieved is 98.97%. It is evident from **Figure 3(a)** that the model neither overfits nor underfits the provided data.



(a)



(b)



(c)

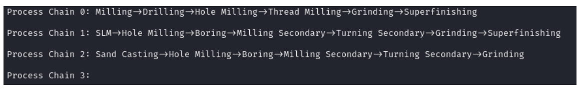
Figure 3 - The learning curves for a system with 128 LSTM cells, trained for 10 epochs. The loss is measured using the categorical cross-entropy loss function.

Subsequently (though this step was technically performed before training), two samples from the validation set (the first two) were extracted and used as testing samples to evaluate the system's real-world performance. The input and expected output for these samples are as follows.

Table 4 - The testing input and expected output.

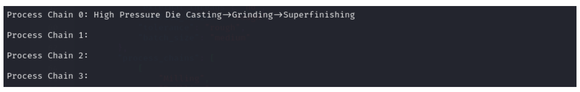
Input 1: "geometry" : "prismatic", "holes" : "large_functional", "external_threads" : "no", "surface_finish" : "very_good", “tolerance" : "tight", "batch_size" : "small"	***Input 2:*** "geometry" : "axisymmetric_with_prismatic_features", "holes" : "none", "external_threads" : "no", "surface_finish" : "very_good", "tolerance" : "tight", "batch_size" : "mass"
Expected Output 1: **Chain 1** : “Milling”, “Drilling”, “Boring”, “Grinding”, “Superfinishing” **Chain 2** : “SLM”, “Hole Milling”, “Boring”, “Milling Secondary”, “Grinding”, “Superfinishing” **Chain 3** : “Sand Casting”, “Hole Milling”, “Boring”, “Milling Secondary”, “Grinding”, “Superfinishing”	***Expected Output 2:*** **Chain 1**: “High Pressure Die Casting”, “Grinding”, “Superfinishing”

The results of the system are depicted in **Figure 4**. As we can see, the first input results are not exactly matching. Boring in the first chain is replaced with hole milling and thread milling. In the second chain after milling the secondary process the turning secondary is added which obviously is a fault and in the last the last process of superfinishing is not added. The reason for which these results are displayed is not to lose any grades (:)) but rather to demonstrate that although we achieved almost perfect accuracy (~99%) the human factor is really important in determining whether or not the results from an automated system are valid or not!



```
Process Chain 0: Milling→Drilling→Hole Milling→Thread Milling→Grinding→Superfinishing

Process Chain 1: SLM→Hole Milling→Boring→Milling Secondary→Turning Secondary→Grinding→Superfinishing

Process Chain 2: Sand Casting→Hole Milling→Boring→Milling Secondary→Turning Secondary→Grinding

Process Chain 3:
```

(a)



```
Process Chain 0: High Pressure Die Casting→Grinding→Superfinishing

Process Chain 1:

Process Chain 2:

Process Chain 3:
```

(b)

Figure 4 - The prediction of the system for the previously mentioned inputs.