Python vs Julia: Implementing Long Short-Term Memory Neural Networks to Forecast Inflation

Francisco Jose Manjon Cabeza Garcia
February 2, 2023

Abstract[1]
We present the implementation of a long short-term neural network to forecast monthly inflation in the US using two different programming languages: Python and Julia. We compare the implementations with respect to computational running time, accuracy in the predictions and difficulty to code.

[1]This term paper is based on a presentation I held for the seminar *Recent Advances in Computational Macroeconomics* at the TU Berlin in the summer semester 2022 about the paper *MatLab, Python, Julia: What to Choose in Economics* by Chase Coleman, Spencer Lyon, Lilia Maliar and Serguei Maliar [4]

1 Introduction

In [4], the authors give an overview of the advantages and disadvantages of using MatLab, Python and Julia for economics research. The authors also provide running time and accuracy comparisons between the three programming languages using different standard econometrics optimisation models. The information provided there is useful for researchers to decide which programming language they should learn and use, and the authors also provide implementations for the benchmark optimisation models that they use to compare the programming languages. However, there has been a recent rise of machine learning and big data, partially helped by the reduction in the cost of processing power and the easy access to data and machine learning software, which researchers could capitalise on. Researchers who want to apply machine learning and big data methods might be confronted by the fact that the software and programming language that they have been using until now might not be suitable for the new tasks and modelling requirements. Moreover, they might also question if the software that they are using now is also the best with respect to running time, ease of use and capabilities for their modelling tasks.

In this term paper, we present the implementation of a long short-term neural network (LSTM-NN) to forecast monthly inflation in the US using two different programming languages: Python and Julia. We compare the implementations with respect to computational running time, accuracy in the predictions and difficulty to code.

Forecasting inflation has always been one of the main goals of macroeconomists. Inflation is a macroeconomic measure that influences many economic decisions: from monetary policy to fiscal policy as well as microeconomic decisions at company level. While there are many different methods to forecast inflation, like timeseries analyses or factor models, the recent advances in machine learning, and more importantly the substantial drop in processing power and memory costs, allow economists to use more complex models. These models, like neural networks, have the potential to identify non-linear, highly complex patterns and structures in data, which can lead to better forecasts. Improvements in forecasting cannot only be made in terms of accuracy, but also by being able to forecast further into the future and even nowcast [12].

This work is divided into 5 sections including this introduction. In the next section, we provide an overview of how neural networks and in particular LSTM-NN are defined and why they might be a good choice when forecasting inflation. In the third section, we define the benchmark model that we are going to use to compare its implementation between Python and Julia. In the fourth section, we discuss the advantages, disadvantages as well as the implementation differences between the two programming languages. Moreover, we also provide comparisons with respect to running time and prediction accuracy. In the final section, we conclude with a summary of our results and some ideas for future work.

2 Long Short-Term Memory Neural Networks

In this section, we provide a very short overview of how neural networks and in particular long short-term memory neurons work.

Artificial neural networks are mathematical models inspired by the way neurons in the human brain work: neurons transform the sensory input they receive into an output signal in a non-linear manner, and can adapt this non-linear mapping to the inputs they receive, i.e., can learn from their input. In [20], the authors mention that the main advantage of this approach is that neural networks use linear discriminants but in a space where the inputs have been mapped non-linearly, which allow them to learn the non-linearity of the training data with simple algorithms.

We describe how a neural networks work with the following example.

Example 2.1 *Let $a_1, \dots, a_5 \in \mathbb{R}$ be the inputs of a neuron, $b \in \mathbb{R}$ the bias, $w_1, \dots, w_5 \in \mathbb{R}$ the weights of each input and $g : \mathbb{R} \to \mathbb{R}$ a non-linear function, then the output $\tilde{a}$ of the neuron is defined as*

$$\tilde{a} := g(z), \quad \textit{where} \quad z := \sum_{i=1}^{5} w_i a_i + b$$

Figure 1 gives a visual representation of the neuron we have defined inside a dashed box.

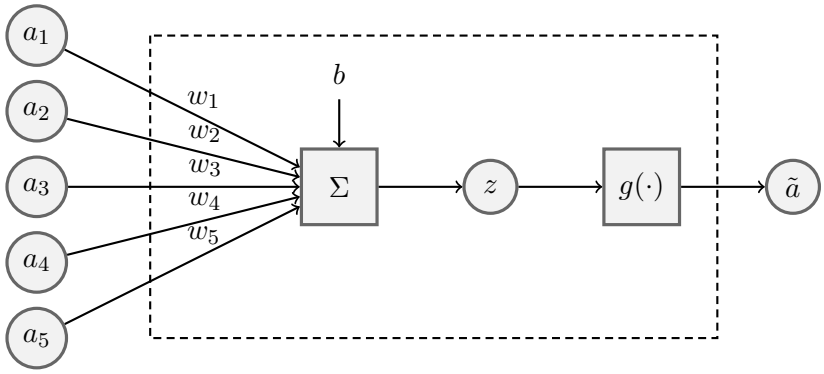

Figure 1: Neuron with 5 Inputs.

Here, we can define some terms as in [20]:

- *The set of inputs $a_1, \dots, a_5$ is called **input layer**.*
- *Each of the weights or connections $w_1, \dots, w_5$ is called **synapse**.*
- *The sum of weighted inputs and the bias z is called **net activation**.*
- *The non-linear function g is called **transfer function**. Examples of commonly used transfer functions are threshold functions, logistic sigmoids, hyperbolic tangent and rectified linear units.*

A neural network is a collection of neurons as in example 2.1 which are interconnected in layers, such that the output of the neurons in one layer are used as input for the neurons in the next layer. An example of a (fully connected) neural network with 3 inputs (a_1, a_2, a_3), 2 hidden layers and one output $(\tilde{a})$ is depicted in figure 2. The last neuron in the network aggregates the outputs of the second hidden layer before returning $\tilde{a}$.

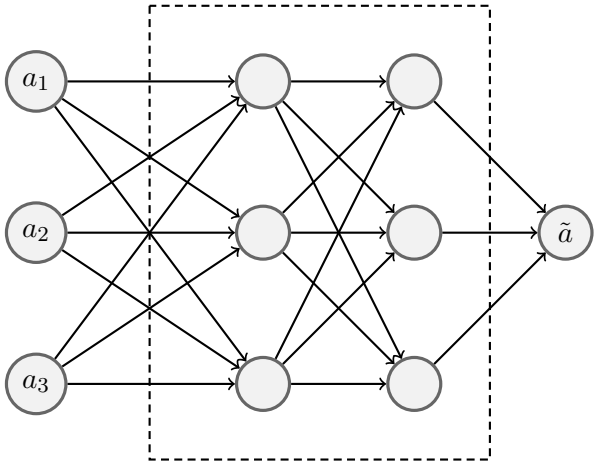

Figure 2: Neural Network with 3 Inputs, 2 Hidden Layers and 1 Output Cell.

Interestingly, a two-layer network is already a quite powerful model to approximate non-linear functions. A proof for that was given by Bishop in [3], where he proved that two-layer networks with sufficiently many neurons and sigmoidal transfer functions can approximate non-linear functions to arbitrary accuracy. However, in practise it is not intelligent to model with two-layered networks only. Specially in economics, the function that we want to approximate or learn can be very high-dimensional and highly non-linear. This means, that more complex neural network structures are necessary, like combining layers with different activation functions, cell count, cell connections (e.g. dropout layers to avoid overfitting), etc. Some authors have presented mixed results for the task of modelling inflation with feed-forward neural networks, for example [16].

One useful neural network structure are recurrent neural networks. These are neural networks where the cells can be connected in cycles, i.e., the data does not only flow forward but can also flow backwards. The output of a cell in a hidden layer can be used as input for the cells in a previous layer. They are particularly useful for economic modelling because they allow to model temporal and dynamic features of the input data. Some authors have experimented with recurrent neural networks to forecast inflation with mixed results too, for example [30]. The most basic recurrent structure is the fully recurrent neural network, where the output of all neurons is used as input for all neurons too. However, one powerful structure, specially for timeseries modelling, was first introduced by Hochreiter and Schmidhuber in [13]: the Long Short-Term Memory (LSTM) network. They solve one of the main disadvantages of tradition recurrent neural networks mentioned in [13] and [18]: vanishing gradients, which is very well summarised in [18] as follows.

> *Suppose that we seek to estimate an RNN with number of lags L. In brief terms, the estimation of the RNN involves computing the gradient of the loss function with respect to the parameters, which implies evaluating the gradient at every time step within sequences of observations of length L. If the parameters are significantly small (usually they are close to zero), the higher L, the smaller the contribution of observations sufficiently back in time, given the multiplicative effect of the chain rule and the fact that the derivative of the activation function is bounded by 1 (supposing the commonly used hyperbolic tangent or sigmoid functions). In other words, the model will not properly estimate long-term dependencies because the estimation process is compromised for sufficiently long sequences.*

LSTM cells tackle this problem by saving an internal memory state. Their structure

and components is depicted in figure 3.

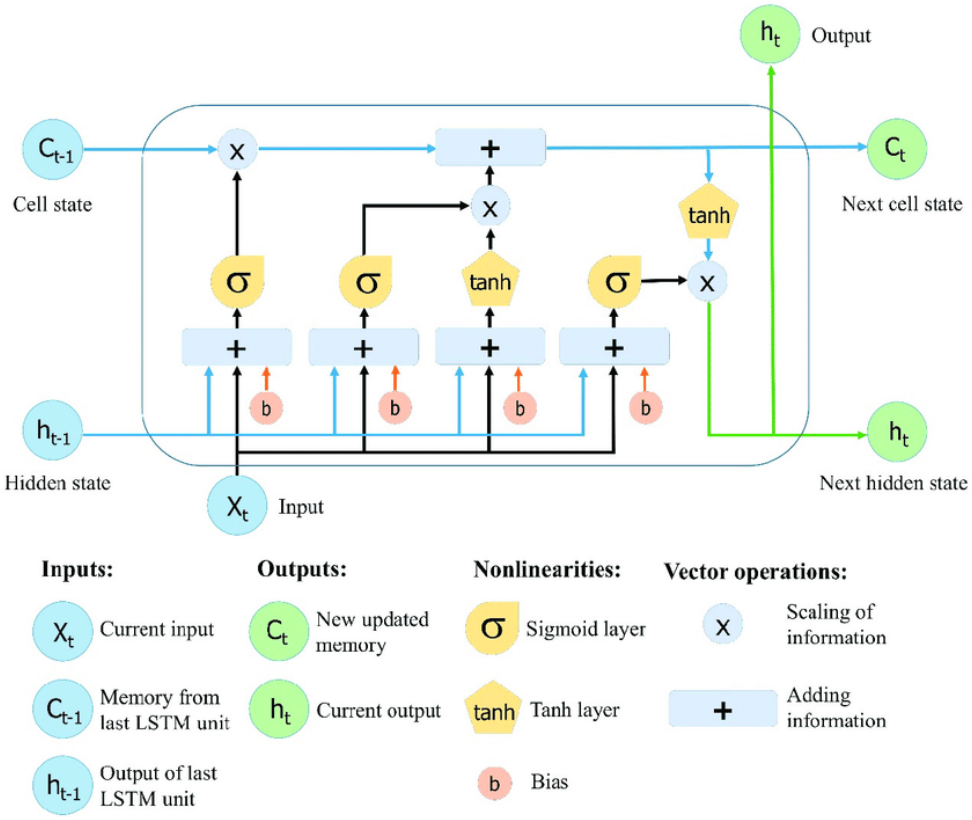

Figure 3: Structure of a LSTM Cell [15]

The output of a LSTM cell is mathematically defined using the notation from figure 3 and as defined in [15] as $h_t := O_t \tanh(C_t)$, where

$$O_t := \sigma(W_0[h_{t-1}, X_t] + b_0),$$
$$C_t := C_{t-1}\sigma(W_f[h_{t-1}, X_t] + b_f) + \tanh(W_n[h_{t-1}, X_t] + b_n)\sigma(W_i[h_{t-1}, X_t] + b_i),$$

W_0, W_f, W_n, W_i are the weight matrices and b_0, b_f, b_n, b_i are the biases of the output gate, forget gate and cell states respectively and σ is the sigmoid function.

In this section, we have provided a very simplified and short description of how neural networks are defined, and how LSTM works, as well as their advantages. We recommend checking all the sources cited in this section for more details about neural networks and LSTM cells in particular. We refrain from providing more details because our focus is on comparing programming languages and not on the mathematics of neural networks, which require deeper mathematical knowledge that we cannot provide in a short term paper like this one.

3 Benchmark Model Specification

In this section, we define the machine learning model that we will use as implementation task to compare the Python and Julia programs. Details about the model structure, training method, etc. are all described here. Details about the input data used for training and predicting are compiled in appendix A.

Finding out which data sources have the best predictive power or building an accurate model are not part of our objective here. We only focus on comparing programming languages. Therefore, we use a benchmark model which is based on and very similar to one proposed by Livia Paranhos in [18]. More specifically the *LSTM-FF* model presented by Paranhos. This is a neural network model composed by an first LSTM-layer, whose output is fed to a feed-forward fully connected neural network as depicted in figure 4.

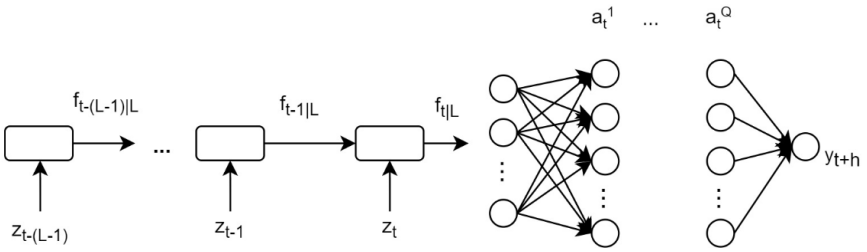

Figure 4: LSTM Model Structure up to L Lags Connected to Feed-forward Network [18].

Our aim is to forecast monthly inflation in the USA 3 months in advance[2] using data from the FRED database. A comprehensive list of the timeseries used as input data and the adjustments and transformations applied to them can be found in appendix A. We use data from 1st January 1990 to 26rd September 2022.

Our specific model is defined as follows:

1. An initial input layer that takes as input 45 timeseries of length 48. That means, out model takes as input 48 lagged consecutive observations of each of the 45 features that we consider. More information about the input data and the features that we used is provided in appendix A. In the model from figure 4, these two values correspond to $L := 48$ and each vector $z_i \in \mathbb{R}^45$ for $i = t - (L-1), \ldots, t$ and $t \in \mathbb{N}$.

2. A first hidden layer with 2 LSTM units. In the model from figure 4, these parameters correspond to $f_{t|L}$-size $p := 2$. The activation function for the neurons in this layers is set to the hyperbolic tangent. Normally, LSTM unit return a prediction for each time step fed into it. That means, that in our case, each LSTM unit would return 48 predictions for each feature in each input element. This corresponds to the prediction that each LSTM unit makes for each sequence of length $1, 2, 3, \ldots, L$. We only keep the last prediction, which corresponds to the prediction based on the whole sequence fed as input, i.e., the sequence of length 48 for each feature. This last prediction for each of the features is then fed into the next hidden layers.

3. The first hidden layer is then connected to a fully connected feed-forward neural network consisting of 4 layers, each with 128 neurons and with a rectified linear activation function. In the model from figure 4, these two values correspond to $Q := 4$ and $n := 128$.

[2] In the model from figure 4, this corresponds to $h := 3$.

4. Finally, the last layer of the feed-forward neural network is connected to an output layer consisting of a single neuron with a linear activation function. The output of this last neuron is the prediction for a given input to the model.

For all units and layers, we use a random number generator following a Glorot uniform distribution, also called Xavier uniform distribution [10], to initialise the weight matrices and zero vectors to initialise the biases.

Before training the model, we prepare our input data as follows:

1. We take all observations for all features up to the point in time at which we retrain the model, and remove the last 3 observations of each feature. We remove the last 3 observations to avoid forward looking bias[3].

2. For each feature, we compute the maximum absolute value of the selected observations, and divide all those observations by that computed factor. This normalises the data, so that all observations lie in the interval $[-1, 1]$.

3. We create sequences of length 48 for each feature with sequence stride 1, sampling rate 1 and without shuffling[4]. That means that for example, if we only have 49 available observations for each feature, we would get 2 sequences for each feature. The first one containing the first 48 observations, and the second one containing the last 48 observations. That means that two consecutive sequences share 47 observations: the first one contains one observation that is not in the following (its first observation), and the second contains one that is not in the first (its last observation). For each matrix of observations $Z \in \mathbb{R}^{48\times 45}$, we assign a target value (label) corresponding to the monthly inflation 3 months in advance from the last point in time of the observations in the matrix.

4. Finally, we split the set of tuples (observations matrix, label) in two. The first one containing the first 75% of the tuples, which we will use a training sample, and the second containing the final 25% of the tuples, which we will use a validation sample.

Initially, we train the model after a warm-up period of 191 months. That means, that the first training point in time corresponds to the end of December 2006. We train the model at the end of each year using the data available until the end of December of that year[5]. For each prediction, we use as model input the last 48 observations available at the end of that month for each feature, and normalise them using the normalisation factors computed in the last retraining step.

To train the model, we use an Adam optimiser mostly with default parameters [14]: $\alpha := 0.001$, $\beta_1 := 0.9$, $\beta_2 := 0.999$ and $\epsilon := 10^{-7}$. Moreover, we measure the accuracy of predictions while training with the mean squared error function. To avoid overfitting and improve the model generalisation, we implement two callbacks for training: one to save the model parameters the validation loss reaches a new minimum during training, and one to stop the training process earlier if the validation loss does not improve by at least 0.001 for more than 3 consecutive epochs. In total, we train our model for 400 epochs, or less

[3]We will try to predict monthly inflation 3 months in advance. Hence, at point in time $t \in \mathbb{N}$, the latest monthly data point that we assume to have is the monthly inflation of that month. To train the model, we need however observations to use as labels that are shifted 3 by months. For example, in September 2010, the last set of inputs should be labelled using December 2010 monthly inflation, but this data is not available in September 2010 yet.

[4]Because we are dealing with timeseries data, we do not shuffle the observations to obtain more samples.

[5]Actually, we only use the data until the end of September of that year, because we do not have labels for the last 3 months, since we are forecasting 3 months in advance.

if the training process is stopped by the callback earlier. After training, we load the last saved parameters, which correspond to the model parameters with the lowest validation loss.

In this section, we have described the benchmark model that we will use to compare its implementation in different programming languages. Our benchmark model has a wide room for improvement. We have chosen most parameters based on the results presented in [18]. However, neither the model structure nor the model parameters have been subject to optimisation. Hence, we do not expect the model to deliver predictions worth to be used in real-world applications or to discuss in an academic paper. As already mentioned at the beginning of this section, our model is only intended to be a tool to compare programming languages.

4 Results

In this section, we discuss the differences in code for each of the implementations, as well as their performance with respect to running time and prediction accuracy.

In [4], a comprehensive list is provided with the advantages and disadvantages of Python and Julia for economics research and modelling. Hence, we focus on Python's and Julia's advantages, disadvantages and differences for machine learning modelling. More specifically for neural network modelling and training.

Developing machine learning models is an iterative process, because models normally need to be tweaked before they achieve their goal. It is not like implementing optimisation methods to solve classical econometric problems, which can be solved by implementing a solver. Machine learning models are implemented by trial and error and by comparing the performance and requirements of different methods as well as different hyperparameters and structural features of the model. Here, both programming languages have some features that make them specially suitable for the task.

- They can be used together with Jupyter Notebooks. Jupyter Notebooks are a web-based platform to run code interactively. It supports Python as well as Julia code. This tool is very helpful for iterative development, because you can quickly make changes to code and run it in pieces, without having to run a whole program to test the changes.
- They are high-level programming languages, which flatten the learning curve for people learning to code with them.
- As already mentioned in [4], they are free programming languages. This reduces the cost of using it compared to proprietary languages like MatLab, and also has the side effect of having a big open community sharing their work and knowledge online.

Both programs are structured in 6 blocks:

1. User parameters. Here the user can modify the most important parameters to run the script.
2. Import packages and methods. Here the necessary external packages and methods are imported.
3. Import and prepare raw data. Here the input data, which also contains the target variable data, is read.
4. Define methods to get the neural network model. In the Python implementation, only one method is defined in this section, which generates and compiles an untrained instance of the benchmark model from section 3. In the Julia implementation, two methods are defined to generate the untrained benchmark model and to train it according to the specifications, i.e., including the callbacks.
5. Create a model and train it at every retraining point. Here the model training and prediction computations are performed.
6. Plot prediction results and observed target value. Here the prediction vs observed plot is generated and prediction accuracy measures are computed.

We run both programs on a Lenovo IdeaCentre 3 07IMB05 with an Intel Core i5 10400, UHD Graphics 630, 8 GB of RAM and a 512 GB SSD.

4.1 Python Implementation

The Python program is mostly implemented around the `TensorFlow` Python package and the `Keras API` contained in it. In the `Keras API` GitHub repository [23], `TensorFlow` and `Keras API` are defined as follows.

> *`TensorFlow` is an end-to-end, open-source machine learning platform. You can think of it as an infrastructure layer for differentiable programming. It combines four key abilities:*
>
> - *Efficiently executing low-level tensor operations on CPU, GPU, or TPU.*
> - *Computing the gradient of arbitrary differentiable expressions.*
> - *Scaling computation to many devices, such as clusters of hundreds of GPUs.*
> - *Exporting programs ("graphs") to external runtimes such as servers, browsers, mobile and embedded devices.*
>
> *Keras is the high-level API of TensorFlow 2: an approachable, highly-productive interface for solving machine learning problems, with a focus on modern deep learning. It provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity.*

`TensorFlow` is, together with `PyTorch`, one of the most used Python packages for machine-learning. `TensorFlow` was initially developed by the Google Brain Team for internal use in research and production and in 2015 released under the Apache License 2.0 [2]. On the other hand, `PyTorch` was initially developed by Meta AI and later integrated as part of the Linux Foundation umbrella according to [19]. The main difference between the two packages is that `TensorFlow` compiles the computation graphs[6], while `PyTorch` works dynamically by executing them. However, `TensorFlow` has added an *Eager* mode which also allows computation graphs execution dynamically and is activated by default in `TensorFlow 2` [27].

In our Python program, there are some example of how powerful and easy to use is the `TensorFlow` package. After importing the input data with the help of the `read_csv` method from the `Pandas` package, we can find the `GetModel` method in our code. Here, we define each layer of our network and chained them to later define our model. At the definition of each layer, we can easily choose the type of layer from the `Keras API` (in our case `Input`, `Dense` or `LSTM` layer), and `TensorFlow` takes care of the connections between them. Moreover, we can easily choose several parameters for our layers, like the activation function, the kernel and bias initialisation methods or the number of cells. Finally, the model is defined by proving the input and the output layer, and compiled using an Adam optimiser and the mean squared error as error function. In 20 lines, we have defined our model as in section 3, and in 2 compiled it with random initial parameters. We could have even done this in 7 lines if code readability had been ignored. The model object returned by the `GetModel` method is not yet trained, but it can already transform an input into an output according to the neural network model defined in section 3.

The `Keras API` provides access to over 100 different neural network layer types with parameters to adjust their weights and bias initialisation, as well as their activation function, initial state, number of cells, etc. Almost 20 different activation functions are also pre-built, and creating your own activation function is very simple if you know how to

[6] One can think of computational graphs as the structure of the model, which defines how data is processed from input to output.

define a Python method. Around 30 different initialisation functions and random number generators are provided by `Keras`. To compile your model, `Keras` has more than 15 different loss functions and 9 optimisers available. You can find more information in [26].

The model training can also be done in just 1 line of code with the help of `Keras` objects, including the callbacks for early training stop and optimal validation loss parameter selection. `Keras API` provides 15 built-in callbacks, for example, to early stop training, log training internals, save parameters or to modify optimiser parameters if the training process stalls.

Another useful feature in `TensorFlow` are `dataset` objects and the methods to generate and handle them. We use one of those methods to generate the sequences of length 48 automatically from the data timeframes selected for training and validation. In our computation example, generating those sequences is not specially difficult, as one can see in the Julia implementation. However, `dataset` objects and the methods to generate them included in `TensorFlow` are very useful to generate datasets by, for example, shuffling observations, reformatting data or to improve efficiency if we want to handle big amounts of data. The `Keras API` also has a `dataset` class, which gives access to some widely known datasets that are commonly used as benchmarks to train and test models. For example, the MNIST handwritten digits dataset or the Boston housing price regression dataset.

Finally, `TensorFlow` offers support for parallel [25] and GPU[7] [24] computing, which are two of the most important advances in machine learning and big data in last decades. `TensorFlow` automatically uses GPU acceleration to execute certain methods, like the `tf.matmul` method for matrix multiplication, if a GPU is available in the system it is running. Moreover, manual device placement is made very simple: you can just put your code inside a `with tf.device` block. This is very useful because GPUs tend to have more processing units than CPUs, and are optimised for matrix and vector operations that are commonly needed for graphics applications. A more recently implemented and advance feature are TPU[8] support [28], which are processors developed and optimised for machine learning computations. `TensorFlow` supports manual CPU, GPU and TPU device placement as well as distributed strategies for them. This makes the whole modelling easier, because the user can take care of infrastructure questions directly and easily through it, and also increases the speed at which models can be trained and executed. Google even give access to GPU and TPU devices when running Python code and `TensorFlow` on their Google Colab platform [11], which can be considered a cloud Jupyter Notebook platform[9]. Other performance enhancing strategies like `TensorFlow` functions and variables, that avoid typical Python inefficient steps like object mutations and list appends [27].

Apart from `TensorFlow`, our Python program also uses 3 more very well-known Python packages: `NumPy`, `Pandas` and `Matplotlib`. They offer mathematical methods, data manipulation and plotting capabilities respectively. All 4 are mostly written in C, which results in efficient execution, even if our program is written in a rather slow interpreted language like Python. They can be considered Python interfaces to C code libraries.

From the perspective of researchers that are not focused on computational methods and only want to analyse data and model processes, the last remarkable advantage of all Python packages used for this project is that they are very widely used in academics and many industries. This results in an easy and ample access to support, documentation and pre-made solutions that can simplify and speed up research tasks, specially in economics.

You can find the Python code in the GitLab repository of this project at `https://git.tu-berlin.de/fcomanjon/neural-networks-computational-macroeconomics`.

[7] Graphical Processing Unit

[8] Tensor Processing Unit

[9] This is a very simplified description of Google Colab.

4.2 Julia Implementation

The Julia program is mostly implemented around the `Flux` Julia package. The documentation page of this package defines it as follows.

> *Flux is a library for machine learning geared towards high-performance production pipelines. It comes "batteries-included" with many useful tools built in, but also lets you use the full power of the Julia language where you need it. We follow a few key principles:*
>
> - *Doing the obvious thing. Flux has relatively few explicit APIs for features like regularisation or embeddings. Instead, writing down the mathematical form will work – and be fast.*
> - *Extensible by default. Flux is written to be highly extensible and flexible while being performant. Extending Flux is as simple as using your own code as part of the model you want - it is all high-level Julia code. When in doubt, it's well worth looking at the source. If you need something different, you can easily roll your own.*
> - *Performance is key. Flux integrates with high-performance AD tools such as Zygote.jl for generating fast code. Flux optimizes both CPU and GPU performance. Scaling workloads easily to multiple GPUs can be done with the help of Julia's GPU tooling and projects like DaggerFlux.jl.*
> - *Play nicely with others. Flux works well with Julia libraries from data frames and images to differential equation solvers, so you can easily build complex data processing pipelines that integrate Flux models.*

`Flux` is the main Julia package for machine learning applications. There are other packages like `SciML` and `DiffEqFlux` which provide access to numerical methods for optimisation, but they are rather suitable for mathematical modelling of numerical problems like differential equations, equation solvers or Montecarlo methods, and not for general machine learning modelling like neural networks are in `TensorFlow` for Python. Our Julia program shows some of the capabilities of the `Flux` package, and also highlights the differences between it and `TensorFlow` for Python.

First, our program imports the input data using two Julia packages: `DataFrames`, which is a package similar to `Pandas` for Python, to manipulate data, and `CSV` to read .csv files, which the `DataFrames` package is not capable of by itself. The `CSV` reads the csv file and parses it into a `DataFrame` object.

After the data import, our program defines 2 methods: one to construct the benchmark model from section 3 and one to train it according to the specifications from section 3. While in Python we only have one method to construct and also compile the model and the training was performed in one line of code, this is not possible in Julia. The reason for this is the reduced availability of built-in callbacks to, in our specific case, early stop training if the validation loss does not improve enough for some training epochs and to save and load the model parameters with the lowest validation loss during and after training. In Julia, we had to add a second method to our program to track validation loss during training, implement the early stopping strategy and save and load the model parameters with the lowest validation loss during and after training.

With the `Flux` package, we can define our model in just 16 lines of code, 9 if we ignore code readability. It gives us access to already built-in layer types like the ones we used: `LSTM` and `Dense` layers. Moreover, we do not need to specify any input layer, as it was the case with the `TensorFlow` package for Python. The `Flux` package comes with many

types of layers built-in [8]. However, not as many as `TensorFlow` and with less parameters to adjust them to our needs. For example, in our program, we had to define a layer between the LSTM and the first dense layer to only consider the last sequence element of the LSTM layer output. In Python, we did this by simply passing the argument `return_sequences=False` to the `Keras API` layer class `LSTM`. Moreover, the `Dense` layer in `Flux` does not allow us to pass an initialiser for the bias, contrary to what was possible with the `Dense` layer in `Keras`. Fortunately, our benchmark model specifies that biases are initialised with zeros, which is the default bias initialisation for `Dense` layer in `Flux`. Regarding activation and parameter intialisation functions, `Flux` provides enough different ones already built-in. However, they are not as many as the ones available in `TensorFlow` for Python.

Once our model is defined with `Flux`, it does not need to be compiled with an optimiser and a loss function before training, as it was the case with our `TensorFlow` model. These two are only passed to the model training method instead. For our program, we had to write our own model training method, because the training callbacks provided by `Flux` were not enough to implement the early stopping strategy and optimal parameter selection method defined in section 3. Our training method takes care of that by computing and tracking the validation loss, as well as save and load the model parameters with the lowest validation loss during and after training respectively. This last point is also a source of differences between `Flux` and `TensorFlow`: while `TensorFlow` has built-in capabilities to export and import model parameters and complete models, `Flux` relies on another package to do that. The package is called `BSON`, and it is a Julia package commonly used to import and export Julia objects and data. We had to import this package in our program to be able to import and export our model. Here also another difference: while in our Python program we only export and import model parameters, in our Julia program we export and import the whole model. This has the disadvantage that it takes longer to import and export. However, it also makes sure that we import the correct model structure when parsing and import into an already defined model object. Moreover, it exports the whole model structure with parameters every time we save it, avoiding the problem of accidentally having saved the model parameters but not having the model structure to run the model again later.

With respect to GPU acceleration, `Flux` supports it out of the box if CUDA and CUDNN are installed [7]. It achieves it by leveraging the `CUDA` Julia package, which allows Julia code to run on NVIDIA GPUs. However, it does not support TPUs like `TensorFlow` does.

Finally, we coded the data preparation ourselves for the training, validation and test datasets in our project. In contrast to the `TensorFlow` package and the `Keras API` features for dataset construction, `Flux` does not provide methods and objects to deal with the data preparation process. However, the `Flux` package supports and can easily handle data prepared by another Julia package called `MLUtils` [9]. We refrained from using the `MLUtils` package because the preparation of our training, validation and test data is relatively simple. While `TensorFlow` and the `Keras API` made our simple data preparation task even easier, the Julia implementation achieves it in just a couple of extra lines of code and the help of Julia `Vector` objects and the `DataFrames` package. Meticulous code readers will also notice that the input data for our model, as well as its output is stored in objects like `Matrix{Float32}` or `Vector{Float32}`, i.e., not using the highest precision which modern processors can handle (64 bits floating point numbers). This is a well known limitation of the `Flux` package, which cannot handle objects of type `Float64` together with other number object types [22]. This reduces the precision in the computations, but at the same time speeds them up too. Such a limitation must be taken into account, even

if we do not need the extra precision of 64 bit compared to 32 bit floating point numbers.

We conclude that `Flux` provides more than enough built-in features and capabilities to create machine learning models, specially for economists who are not very concerned about the small details like the initialisation parameters of certain layers following an uncommon method. For researchers that want to get deeper into the construction of machine learning models, `Flux` provides the possibility of defining your own methods and objects to pass to `Flux` objects and methods, so that you can create models exactly as you want. This is specially made possible by the fact that the whole `Flux` package is itself written in Julia, and the classes, methods and objects it works with can be defined in Julia. An example of this is our program, where we have implemented callbacks for an early training stopping strategy and optimal parameter selection which are not contained in the `Flux` package. Unfortunately, inexperienced users might find Julia more challenging because it is a much younger programming language and ecosystem than Python, as already mentioned in [4]. Hence, help and documentation is more scarce. Specially for specialised and niche packages like `Flux` or others mentioned here.

You can find the Julia code in the GitLab repository of this project at `https://git.tu-berlin.de/fcomanjon/neural-networks-computational-macroeconomics`.

Float32 vs Float64 with Flux

4.3 Running Time Comparison

To measure the running time of each implementation, we use a Windows PowerShell script[10], which runs each script 10 times, and measures its execution time with the help of the stopwatch feature in `system.diagnostics.stopwatch`. Each implementation is run in a separate script, and we run each script 10 times to avoid results that are skewed due to the processor warm up state. The execution time of each run is measured in milliseconds. We have compiled the best, worst and average running times of each implementation in table 1 rounded to one decimal place. The execution time of each run can be found in the GitLab repository of this project.

Programming Language	Best	Worst	Average
Python	59620.6	66404.8	60882.6
Julia	99068.4	105426.8	100414.1

Table 1: Running Time Statistics in Milliseconds for Each Programming Language.

The results in table 1 show a behaviour that some would consider unexpected: the Python implementation runs on average in 40% less time than the Julia implementation. It seems that the Julia-written `Flux` package and Julia's just-in-time compilation cannot hold against highly optimised and mostly in C written `TensorFlow` Python package.

Moreover, we believe that the maturity of the Python packages used is substantially greater than that of the corresponding Julia packages. For example, while the development of the `Pandas` Python package started in 2008 and its first open source release happened in 2009 [17], the `CSV` Julia package initial release happened in 2015 [1]. There were and there are also big companies behind the initial development and current development support of Python packages, like the hedge fund company AQR Capital Management, which developed the initial version of `Pandas` [17], or Google, which develops the `TensorFlow` package [2].

[10]The Windows PowerShell scripts are also available in the GitLab repository: `https://git.tu-berlin.de/fcomanjon/neural-networks-computational-macroeconomics`.

4.4 Prediction Accuracy Comparison

The runs performed to measure the execution time of the Python and Julia programs also provided data about the prediction accuracy of the resulting trained models. In table 2, the best, worst and average mean squared error, mean absolute error and maximum absolute error are summarised.

	Python			Julia		
Statistical Metric	**Best**	**Worst**	**Average**	**Best**	**Worst**	**Average**
MSE	0.00003	0.00010	0.000057	0.00015	0.00065	0.000393
MAE	0.00415	0.00799	0.005762	0.00881	0.01820	0.013598
MaxAE	0.01679	0.04027	0.025986	0.04900	0.09584	0.065461

Table 2: Prediction Accuracy Statistics for Each Programming Language.

With respect to prediction accuracy, we observe an underperformance of the Julia implementation compared to the Python implementation across the board. With respect to the mean squared error, which was the loss function that we considered for the model training, the Julia program achieves an average error almost 7 times higher than the corresponding for Python. For the mean absolute error, the Julia program is around twice as bad as the Python program on average. The maximum absolute error measures the maximum absolute difference between the prediction and the actual observation, and is also more than twice as high in the Julia average run compared to the Python average run.

To provide some visual results, we have selected one run of each program. Figures 5 and 6 show the line plot of the prediction and actual observations of the best Python and Julia run with respect to mean squared error from the running time analyses respectively. The Python run corresponds to the second of the 10 runs. It took 61234.4 ms to execute, and delivered predictions with a mean squared error of 0.00003, mean absolute error of 0.00415 and maximum absolute error of 0.01679. The Julia run also corresponds to the second of the 10 runs. It took 100045.99 ms to execute, and delivered predictions with a mean squared error of 0.00015, mean absolute error of 0.00881 and maximum absolute error of 0.051721.

As we can see, the Python predictions appear to follow the general trend of the actual observations, but fails to provide predictions that are accurate enough to be used in real-world applications. Interestingly, the model seems to track the first drop in monthly inflation around the Great Financial Crisis around 2008, but only predict the second drop late, when actual observed inflation had already turned back positive. The actual observed trough short after the start of the year 2020 shows one of the biggest problems researchers and practitioners face when modelling economic timeseries and events. The start of the COVID-19 pandemic produced economic indicator readings which were completely unusual. This affects models that are trained with short time windows worth of data. For our model, this resulted in excessively mild predictions of inflation (in fact of deflation) than the inflation figures that actually materialised. Machine learning models are particularly susceptible to these effects, because their success is highly dependent on the amount and quality of the input data used to train, validate and test the models. Nevertheless, the period after the start of the COVID-19 pandemic and before the start of the most recent rise in inflation is the best performing period of the model, with predictions almost perfectly matching the observed inflation readings. This performance however falls as soon as the latest inflationary period started, when the model has been predicting deflation and inflation has kept rising.

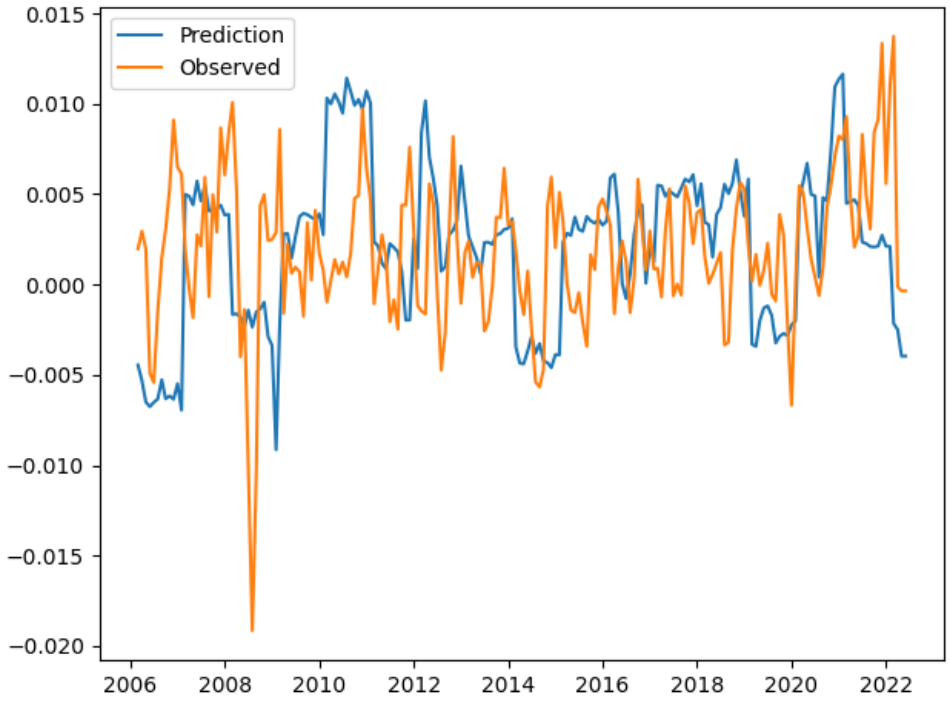

Figure 5: Prediction vs Observations from The Second Python Implementation Run.

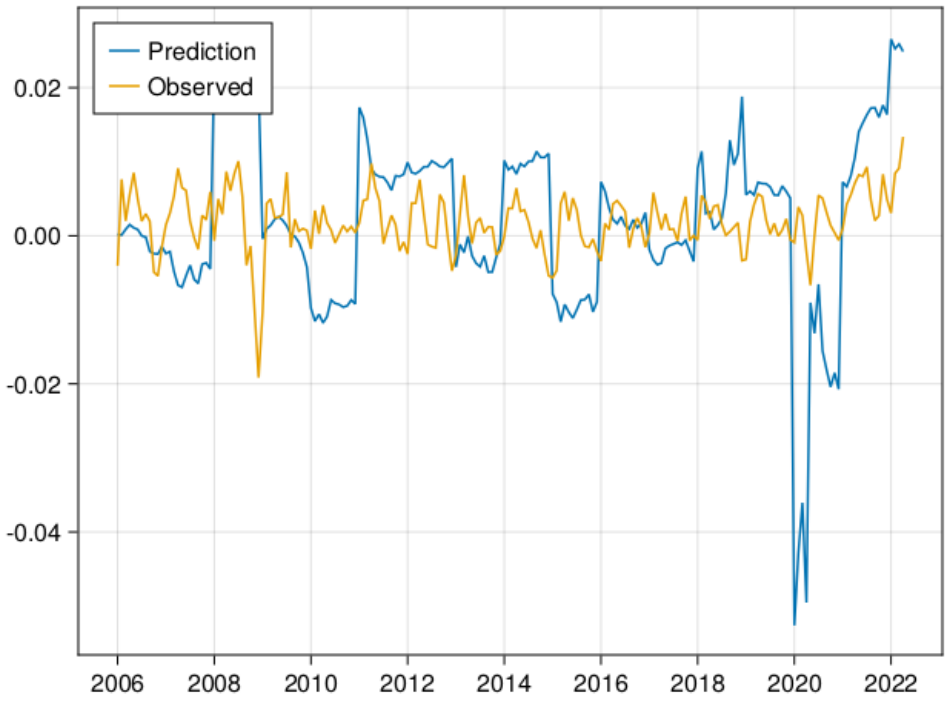

Figure 6: Prediction vs Observations from The Second Julia Implementation Run.

The Julia predictions plot looks considerably worse than the Python plot. There are very few periods during which the predictions seems to at least track the trend of the actual observations. If any, these periods are also very short. The overshoots to

predictions of monthly inflation around -4% are a concern if this model was to be applied in the real-world. Again, this inaccurate predictions coincide with the start of the COVID-19 pandemic, which resulted in extreme economic data readings for which the model has not been trained. Hence, the extreme predictions of deflation, which did not materialise in reality.

The prediction accuracy results show once again that neither our model nor our data are optimised for the task of predicting monthly inflation in the USA 3 months in advance. However, they help to compare the performance of the two programming languages and their respective machine learning packages.

In this section we have discussed the machine learning packages used for our Python and Julia implementations. We have provided information about their advantages, disadvantages and potential. Moreover, we have compared both implementations with respect to their execution time and the resulting predictions' accuracy. In general, we can conclude that the Python program is substantially better than the Julia program in all aspects analysed. This highlights the massive advantages of the `TensorFlow` Python package compared to the `Flux` Julia package.

5 Conclusion

In this section, we conclude this term paper in which we have shortly defined how neural networks and LSTM cells work in section 2. We have defined a benchmark model in section 3 following the one defined in [18]. Finally, we have compared the model implementation in Python with the one in Julia in section 4.

We conclude that both Python and Julia offer machine learning packages capable of modelling neural networks with great flexibility with respect to the network structure, training and testing. Both `TensorFlow` and `Flux` come with many features already built-in. These built-in features should be more than enough to start modelling problems with neural networks and get accurate results. The differences appear once we want to get into deeper details like training strategies (e.g. early stopping), where the maturity of `TensorFlow` and its backing by Google can be noticed.

The `TensorFlow` is very complete package, which can help throughout the whole modelling process. It has features to deal with the data preparation step, the model definition including very specific details, like parameter initialisation, and training in distributed systems with processing units specific for machine learning (TPUs). These features make the creation of efficient and powerful machine learning models very easy by leveraging high level programming and abstraction. `TensorFlow` can be consider the best of two worlds: programmers get to code using a very simple programming language like Python while profiting from the efficiency and power of a complex programming language like C.

`Flux` is not as complete as `TensorFlow`, but offers enough features to deal with most modelling problems that researchers face, specially in economics. The main advantage of this Julia package is the fact that it is written in Julia. Hence, if some feature is not available, it can be easily be implemented by users who are familiar with the Julia programming language. Adding or changing parts of `TensorFlow` is not as easy if you are not familiar with the C programming language or if `TensorFlow` does not offer a direct option through the parameters of its objects and methods. This comes however with a cost: even Julia's just-in-time compilation feature cannot hold against Python packages like `TensorFlow` that are mostly written in C. Although just-in-time compilation is in general a clear efficiency booster compared to an interpreted language like Python for programs that are computationally heavy.

Both `TensorFlow` and `Flux` are open source packages, and both Python and Julia are free programming languages. However, the backing behind `TensorFlow` by Google and behind other Python packages is, together with their longer lifetime, a clear advantage with respect to available functionalities, documentation and examples compared to a much younger language like Julia. Nevertheless, the Julia ecosystem is evolving and growing fast. For our project, we could find enough information to build our program, even though we did not have previous experience with the `Flux` package. One can also confirm this observation on the GitHub repositories of the packages used for this project, where users are in constant exchange and new functionalities are quickly implemented.

Our initial aim for this term paper was to compare 3 programming languages: Python, Julia and C#. The first is an interpreted language which is very popular and easy to learn, the second is a just-in-time compile language which is gaining popularity fast and the third is a compiled programming language which is easier to learn than C but much more efficient than the first two. However, we could not find any machine learning library for C# which matched the capabilities of `TensorFlow` and `Flux`. Hence, we would have had to code many parts ourselves, which would not have set C# in a good starting position, because we are no match to the programmers working on the `TensorFlow` and `Flux`. The same also applies to MatLab, the third programming language analysed in [4]. We do also

believe, that researchers are not interested in using programming languages that increase their workload and require special skills to achieve what `TensorFlow` and `Flux` can do in just a couple of code lines.

Another set of analyses that we did not performed was running our code on GPUs or TPUs. Unfortunately, we do have access to any computer with a GPU powerful enough to match the processing power of the CPU we used to run our code. An alternative could be renting a cloud virtual machine with GPU, but this was outside of our expertise field and budget. Google Colab [11] gives free access to a GPU or a TPU to run Python code on their cloud-based Jupyter Notebook. However, this platform is optimised to run Python code, which would have set Julia in a disadvantageous position to start with. Because of this and the fact that we wanted to compare the two, we refrained from running our code on Google Colab's GPU or TPU instances. Nevertheless, we recommend using the Google Colab platform if you want to use `TensorFlow` and want to try running the code on a GPU or TPU instance for free. Other platforms do also offer systems to efficiently run and deploy machine learning models, like *Amazon SageMaker* or *Azure AI*. Running `TensorFlow` on a GPU or TPU does increase efficiency considerably compared to CPUs as several users have reported ([5], [21], [29]).

In the end, researchers can get away with inefficiencies which industry professionals might not. Hence, if you do not plan to deploy your machine learning algorithm in a real-world application and only want to analyse data to prove research hypotheses, you can simply choose the programming language which you can work most efficiently and are comfortable with. Researcher's life hours will always be worth more than the cost of buying a better GPU or running your model on a more powerful cloud-based machine to compensate using a slower programming language.

References

[1] GitHub - JuliaData/CSV.jl, 2022. URL https://github.com/JuliaData/CSV.jl.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016. URL http://arxiv.org/abs/1603.04467.

[3] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995. ISBN 0-19-853864-2.

[4] Chase Coleman, Spencer Lyon, Lilia Maliar, and Serguei Maliar. Matlab, Python, Julia: What to Choose in Economics? *Computational Economics*, 58(4):1263–1288, December 2021. doi: 10.1007/s10614-020-09983-. URL https://ideas.repec.org/a/kap/compec/v58y2021i4d10.1007_s10614-020-09983-3.html.

[5] Datamadness. TensorFlow 2 - CPU vs GPU Performance Comparison, 2019. URL https://datamadness.github.io/TensorFlow2-CPU-vs-GPU.

[6] St. Louis FED. Federal Reserve Economic Data Website, 2022. URL https://fred.stlouisfed.org/.

[7] Flux. GPU Support - Flux, 2022. URL https://fluxml.ai/Flux.jl/stable/gpu/.

[8] Flux. Model Reference - Flux, 2022. URL https://fluxml.ai/Flux.jl/stable/models/layers/.

[9] Flux. Working with data using MLUtils.jl - Flux, 2022. URL https://fluxml.ai/Flux.jl/stable/data/mlutils/.

[10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL https://proceedings.mlr.press/v9/glorot10a.html.

[11] Google. Colaboratory, 2022. URL https://colab.research.google.com/.

[12] Giselle Guzmán. Internet search behaviour as an economic forecasting tool: The case of inflation expectations. *Journal of Economic and Social Measurement*, 36:119–167, 2011.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. URL https://arxiv.org/abs/1412.6980.

[15] Xuan Hien Le, Hung Ho, Giha Lee, and Sungho Jung. Application of long short-term memory (lstm) neural network for flood forecasting. *Water*, 11:1387, 07 2019. doi: 10.3390/w11071387.

[16] Emi Nakamura. Inflation forecasting using a neural network. *Economics Letters*, 86: 373–378, 03 2005. doi: 10.1016/j.econlet.2004.09.003.

[17] Inc. NumFOCUS. pandas - Python Data Analysis Library, 2022. URL `https://pandas.pydata.org/about/index.html`.

[18] Livia Paranhos. Predicting inflation with neural networks, 2021. URL `https://arxiv.org/abs/2104.03757`.

[19] Mo Patel. When two trends fuse: PyTorch and recommender systems, 2017. URL `https://www.oreilly.com/content/when-two-trends-fuse-pytorch-and-recommender-systems/`.

[20] Peter E. Hart Richard O. Duda and David G. Stork. *Pattern Classification*. Wiley-Interscience, 2001. ISBN 978-1-118-58600-6.

[21] Purnendu Shukla. Benchmarking CPU And GPU Performance With Tensorflow, 2021. URL `https://www.analyticsvidhya.com/blog/2021/11/benchmarking-cpu-and-gpu-performance-with-tensorflow/`.

[22] Emiko Soroka and Brian Chen. Opaque error caused by Float64 input to RNN #1972, 2022. URL `https://github.com/FluxML/Flux.jl/issues/1972`.

[23] Keras Team. keras-team/keras: Deep Learning for Humans, 2022. URL `https://github.com/keras-team/keras`.

[24] Tensorflow.org. Use a GPU, TensorFlow Core, 2022. URL `https://www.tensorflow.org/guide/gpu`.

[25] Tensorflow.org. Distributed training with TensorFlow, TensorFlow Core, 2022. URL `https://www.tensorflow.org/guide/distributed_training`.

[26] Tensorflow.org. Module: tf - TensorFlow c2.10.0, 2022. URL `https://www.tensorflow.org/api_docs/python/tf`.

[27] Tensorflow.org. Better performance with tf.function, TensorFlow Core, 2022. URL `https://www.tensorflow.org/guide/function`.

[28] Tensorflow.org. Use TPUs, TensorFlow Core, 2022. URL `https://www.tensorflow.org/guide/tpu`.

[29] Thomas Wiemann. GPU vs CPU benchmarks with Flux.jl, 2022. URL `https://thomaswiemann.com/GPU-vs-CPU-benchmarks-with-Flux.jl`.

[30] Tea Šestanović and Josip Arnerić. Can recurrent neural networks predict inflation in euro zone as good as professional forecasters? *Mathematics*, 9(19), 2021. ISSN 2227-7390. doi: 10.3390/math9192486. URL `https://www.mdpi.com/2227-7390/9/19/2486`.

Appendices

A Data

To get some data to use as input for the model and also fetch data for our target variable (monthly inflation rate in the USA), we have used the Federal Reserve Economic Database (FRED). The FRED is a free data base containing 818000 US and international time series from 110 sources [6]. Our aim is to extract relevant information from the set of selected timeseries with the help of a neural network to predict monthly inflation rate in the USA 3 months in advance.

We fetched data starting 1st January 1990 and ending 26rd September 2022. Our target timeframe for all timeseries is monthly, i.e., each observation in each timeseries corresponds to a monthly read of the timeseries. Because some timeseries are provided with a higher resolution, for example weekly or daily, we adjusted them to a monthly resolution by taking the average over the month or the end-of-the-month value only. For all timeseries and months, we took the first reported value, i.e., we did not consider any revisions after their release data to avoid forward-looking bias. In total 45 timeseries were considered.

The following list (alphabetically sorted by FRED series ID) gives a detailed[11] description of each of the timeseries selected, as well as the resolution adjustments and transformations applied, if any. At this point, we would like to remind the reader, that the objective of this term paper is to compare programming languages. Hence, the timeseries selected here might not be the optimal ones to forecast our target variable in a real-world application or for research purposes. The same as for the benchmark model applies here: if you want to improve the accuracy of the predictions for research or real-world applications, we encourage you to see these selection, aggregation and transformation methods critically, and develop your own methods to optimise it before applying the results presented in this term paper.

- ICE BofA US Corporate Index Total Return Index Value. FRED series ID: BAMLCC0A0CMTRIV. This is a daily resolution timeseries, thus we only considered the end-of-month close for each month. We transformed the timeseries to consider percentage changes between each two consecutive months.
- ICE BofA US High Yield Index Total Return Index Value. FRED series ID: BAMLHYH0A0HYM2TRIV. This is a daily resolution timeseries, thus we only considered the end-of-month close for each month. We transformed the timeseries to consider percentage changes between each two consecutive months.
- Monetary Base; Reserve Balances. FRED series ID: BOGMBBM. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- Population Level. FRED series ID: CNP16OV. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- Consumer Price Index for All Urban Consumers: All Items in U.S. City Average. FRED series ID: CPIAUCNS. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive

[11] For more details about the units, seasonal adjustments, etc., please check [6] using the corresponding timeseries FRED ID.

months. The transformed timeseries shifted 3 months is at the same time the timeseries with our target variable (monthly inflation rate in the USA).

- Crude Oil Prices: West Texas Intermediate (WTI) - Cushing, Oklahoma. FRED series ID: DCOILWTICO. This is a daily resolution timeseries, thus we only considered the end-of-month close for each month. We transformed the timeseries to consider percentage changes between each two consecutive months.

- Delinquency Rate on Business Loans, All Commercial Banks. FRED series ID: DRBLACBS. No resolution adjustment was applied. No transformation was applied to this timeseries.

- Delinquency Rate on Consumer Loans, All Commercial Banks. FRED series ID: DRCLACBS. No resolution adjustment was applied. No transformation was applied to this timeseries.

- Delinquency Rate on Loans Secured by Real Estate, All Commercial Banks. FRED series ID: DRSREACBS. No resolution adjustment was applied. No transformation was applied to this timeseries.

- 10-Year Expected Inflation. FRED series ID: EXPINF10YR. No resolution adjustment was applied. No transformation was applied to this timeseries.

- 1-Year Expected Inflation. FRED series ID: EXPINF1YR. No resolution adjustment was applied. No transformation was applied to this timeseries.

- Federal Funds Effective Rate. FRED series ID: FEDFUNDS. No resolution adjustment was applied. No transformation was applied to this timeseries.

- Gross Domestic Product. FRED series ID: GDP. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.

- Real Gross Domestic Product. FRED series ID: GDPC1. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.

- Federal Debt: Total Public Debt. FRED series ID: GFDEBTN. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.

- Bank Credit, All Commercial Banks. FRED series ID: H8B1001NCBCMG. No resolution adjustment was applied. No transformation was applied to this timeseries.

- Commercial and Industrial Loans, All Commercial Banks. FRED series ID: H8B1023NCBCMG. No resolution adjustment was applied. No transformation was applied to this timeseries.

- Real Estate Loans: Residential Real Estate Loans: Revolving Home Equity Loans, All Commercial Banks. FRED series ID: H8B1027NCBCMG. No resolution adjustment was applied. No transformation was applied to this timeseries.

- Deposits, All Commercial Banks. FRED series ID: H8B1058NCBCMG. No resolution adjustment was applied. No transformation was applied to this timeseries.

- Initial Claims. FRED series ID: ICSA. This is a weekly resolution timeseries, thus we took the average over all observations within each month for each month. We transformed the timeseries to consider percentage changes between each two consecutive months.
- Not in Labor Force. FRED series ID: LNS15000000. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- Population Level - 16-19 Years. FRED series ID: LNU00000012. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- Population Level - 20-24 Years. FRED series ID: LNU00000036. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- Population Level - 25-54 Years. FRED series ID: LNU00000060. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- Population Level - 55 Yrs. & over. FRED series ID: LNU00024230. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- M1. FRED series ID: M1SL. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- Velocity of M1 Money Stock. FRED series ID: M1V. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- M2. FRED series ID: M2SL. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- Velocity of M2 Money Stock. FRED series ID: M2V. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- Monetary Base; Currency in Circulation. FRED series ID: MBCURRCIR. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- University of Michigan: Inflation Expectation. FRED series ID: MICH. No resolution adjustment was applied. No transformation was applied to this timeseries.
- 30-Year Fixed Rate Mortgage Average in the United States. FRED series ID: MORTGAGE30US. This is a weekly resolution timeseries, thus we took the average over all observations within each month for each month. No transformation was applied to this timeseries.
- Bank Prime Loan Rate. FRED series ID: MPRIME. No resolution adjustment was applied. No transformation was applied to this timeseries.

- Motor Vehicle Loans Owned and Securitized. FRED series ID: MVLOAS. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- All Employees, Total Nonfarm. FRED series ID: PAYEMS. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- 10-Year Real Interest Rate. FRED series ID: REAINTRATREARAT10Y. No resolution adjustment was applied. No transformation was applied to this timeseries.
- 1-Year Real Interest Rate. FRED series ID: REAINTRATREARAT1YE. No resolution adjustment was applied. No transformation was applied to this timeseries.
- Revolving Consumer Credit Owned and Securitized. FRED series ID: REVOLSL. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- 1-Year Treasury Bill Secondary Market Rate, Discount Basis. FRED series ID: TB1YR. No resolution adjustment was applied. No transformation was applied to this timeseries.
- 3-Month Treasury Bill Secondary Market Rate, Discount Basis. FRED series ID: TB3MS. No resolution adjustment was applied. No transformation was applied to this timeseries.
- Finance Rate on Consumer Installment Loans at Commercial Banks, New Autos 48 Month Loan. FRED series ID: TERMCBAUTO48NS. No resolution adjustment was applied. No transformation was applied to this timeseries.
- Finance Rate on Personal Loans at Commercial Banks, 24 Month Loan. FRED series ID: TERMCBPER24NS. No resolution adjustment was applied. No transformation was applied to this timeseries.
- University of Michigan: Consumer Sentiment. FRED series ID: UMCSENT. No resolution adjustment was applied. We transformed the timeseries to consider percentage changes between each two consecutive months.
- CBOE Volatility Index: VIX. FRED series ID: VIXCLS. This is a daily resolution timeseries, thus we took the average over all observations within each month for each month. No transformation was applied to this timeseries.
- Wilshire 5000 Total Market Full Cap Index. FRED series ID: WILL5000INDFC. This is a daily resolution timeseries, thus we only considered the end-of-month close for each month. We transformed the timeseries to consider percentage changes between each two consecutive months.

B Python Code

```
# This Python script was written by Francisco Jose Manjon Cabeza Garcia (the author) in the Summer
# semester 2022 for the TU Berlin seminar "Recent Advances in Computational Macroeconomics". The script is
# a practical implementation of a benchmark model, which the author used to compare programming languages
# for researchers. The topic of the seminar paper was based on the paper called "MatLab, Python, Julia:
# What to Choose in Economics?" by Chase Coleman, Spencer Lyon, Lilia Maliar and Serguei Maliar. A
# presentation of this paper was delivered for the seminar, and the paper written by the author shows a
# small contribution to the research direction of the original paper mentioned.
#
# This program is structured in 6 blocks:
# 1. User parameters:
#    Here the user can modify the most important parameters to run the script.
# 2. Import packages and methods:
#    Here the necessary external packages and methods are imported.
# 3. Import and prepare raw data:
#    Here the input data, which also contains the target variable data, is read.
# 4. Define a method to get the neural network model:
#    Here a method is defined to generate and compile the untrained benchmark model as specified.
# 5. Create a model and train it at every retraining point:
#    Here the model training and prediction's computations are performed.
# 6. Plot prediction results and observed target value:
#    Here the prediction vs observed plot is generated and prediction accuracy measures are computed.

# ------------------------------------------------------------------------------------------------------------
# User parameters.                                                                                           |
# ------------------------------------------------------------------------------------------------------------

data_csv_path = "C:/Users/labelname/Documents/neural-networks-computational-macroeconomics/Final_data_2022.09.26.csv"
best_param_path = "C:/Users/labelname/Documents/neural-networks-computational-macroeconomics/best_params/"
observed_vs_predicted_plot_path = "C:/Users/labelname/Documents/neural-networks-computational-macroeconomics/Plots/"
h = 3                       # Number of months in advance to forecast.
lags = 48                   # Number of past periods to feed into the LSTM cells.
lstm_memory_units = 2       # Number of LSTM cells in the first hidden layer of the network.
dense_layers = 4            # Number of dense layers after the LSTM layer.
dense_layers_units = 128    # Number of cells in each of the hidden dense layers of the network.
val_data_split = 0.25       # Percentage of available data to use for validation at each training point.
warmup_periods = 191        # Number of periods to first train the model.

# ------------------------------------------------------------------------------------------------------------
# Import packages and methods.                                                                               |
# ------------------------------------------------------------------------------------------------------------

import numpy as np                                  # To handle data.
import pandas as pd                                 # To handle data.

from tensorflow import keras as K                   # ML modelling package.
from tensorflow.keras.layers import LSTM, Dense     # Neural network layers.


from dateutil.relativedelta import relativedelta    # This method allows to add months to datetimes easily.
from datetime import datetime
import matplotlib.pyplot as plt                     # Plotting package.

# ------------------------------------------------------------------------------------------------------------
# Import and prepare raw data.                                                                               |
# ------------------------------------------------------------------------------------------------------------

# Import the data we will use to train the model and make predictions.
all_data = pd.read_csv(data_csv_path,
                       sep=";",
                       header=0,
                       index_col=0,
                       parse_dates=True)
# Remove the observations from the last h months, because we do not have labels for them yet.
data = all_data.iloc[:-h, :]

# ------------------------------------------------------------------------------------------------------------
# Define a method to get the neural network model.                                                           |
# ------------------------------------------------------------------------------------------------------------
```

```
def Get_Model(input_size=128, lags=48, lstm_memory_units=2, dense_layers=4, dense_layers_units=128):
    """If no parameters are given, this method returns a compiled but not yet trained Keras Model object
    with the hyperparameters and model specifications of the optimal LSTM-pool model defined in the paper
    Predicting Inflation with Neural Networks by Livia Paranhos.

    input_size=128          : Number of model features, i.e., data timeseries count
    lags=48                 : Lags
    lstm_memory_units=2     : f_{t|L}-size p
    dense_layers=4          : Layers Q
    dense_layers_units=128  : Nodes n

    LSTM activation function: hyperbolic tangent
    FF-layer activation function: rectified linear unit
    Last FF-layer activation function: linear
    Kernel initialiser: Glorot Uniform
    Bias initialiser: Zeros
    Optimiser: Adam
    Loss function for optimisation: Mean squared error
    """
    inputlayer = K.Input((lags, input_size))
    hiddenlayer = LSTM(lstm_memory_units,
                       activation="tanh",
                       return_sequences=False,
                       kernel_initializer=K.initializers.GlorotUniform(),
                       bias_initializer=K.initializers.Zeros(),
                       name="lstm-layer-1")(inputlayer)
    for i in range(dense_layers):
        hiddenlayer = Dense(dense_layers_units,
                            activation="relu",
                            kernel_initializer=K.initializers.GlorotUniform(),
                            bias_initializer=K.initializers.Zeros(),
                            name="dense-layer"+str(i+1))(hiddenlayer)
    outputlayer = Dense(1,
                        activation="linear",
                        kernel_initializer=K.initializers.GlorotUniform(),
                        bias_initializer=K.initializers.Zeros(),
                        name="output-layer")(hiddenlayer)

    model = K.Model(inputs=inputlayer, outputs=outputlayer)
    model.compile(optimizer=K.optimizers.Adam(),
                  loss=K.losses.MeanSquaredError())

    return model

# ------------------------------------------------------------------------------------------------------------------
# Create a model and train it at every retraining point.                                                           |
# ------------------------------------------------------------------------------------------------------------------

# Create a dictionary to save the predictions index by time.
predictions = dict()
# Define a variable for the normalisation factors of each feature.
normalisation_factors = None
# Define a variable for the LSTM-FF-NN.
model = None

for i in range(warmup_periods, data.shape[0]):
    if data.index[i].month == 12 or model is None:
        # Either a new year worth of new data is available or this is the start of the loop.
        # Retrain model.
        print("{0} - Retraining model...".format(data.index[i]))

        # Get the data available until the i-th index.
        # Do not use the last h data points, because at point i, we do not have the labels for them.
        available_data = data.iloc[:i+1-h,:]
        # Compute normalisation factors for each feature, so that their values lie in [-1,1].
        normalisation_factors = available_data.abs().max(axis=0)
        # Normalise available data.
        available_data = available_data / normalisation_factors

        # Get available labels until the i-th index.
        available_labels = data.loc[:, "CPIAUCNS"].iloc[:i+1]
```

```
            # Shift labels by h.
            available_labels = available_labels.shift(-h).iloc[:-h]

            # Compute the number of time steps to use as training sample. Use the rest as validation sample.
            training_elements = int(available_data.shape[0] * (1-val_data_split))
            # Create a tensorflow.data.Dataset object from training_elements many observations and labels by
            # creating sequences of length lags.
            training_dataset = K.utils.timeseries_dataset_from_array(data=available_data[:training_elements],
                                                                     targets=available_labels[:training_elements],
                                                                     sequence_length=lags,
                                                                     sequence_stride=1,
                                                                     sampling_rate=1,
                                                                     # All data in one batch as in Livia Paranhos.
                                                                     batch_size=999999999,
                                                                     shuffle=False)

            # Create a tensorflow.data.Dataset object from (total_elements - training_elements)-many observations
            # and labels by creating sequences of length lags.
            validation_dataset = K.utils.timeseries_dataset_from_array(data=available_data[training_elements:],
                                                                       targets=available_labels[training_elements:],
                                                                       sequence_length=lags,
                                                                       sequence_stride=1,
                                                                       sampling_rate=1,
                                                                       # All data in one batch as in Livia Paranhos.
                                                                       batch_size=999999999,
                                                                       shuffle=False)

            # Create a new untrained model.
            model = Get_Model(data.shape[1],
                              lags=lags,
                              lstm_memory_units=lstm_memory_units,
                              dense_layers=dense_layers,
                              dense_layers_units=dense_layers_units)

            # Train the new model.
            # The model is trained with 2 callbacks: one to save the model parameters if the validation loss
            # improves, and another one to early stop training if the validation loss does not improve by at
            # least 0.001 for 3 consecutive epochs.
            history = model.fit(training_dataset,
                                validation_data=validation_dataset,
                                epochs=400, # As in Livia Paranhos.
                                verbose=False,
                                callbacks=[K.callbacks.ModelCheckpoint(filepath=best_param_path,
                                                                       monitor="val_loss",
                                                                       save_best_only=True,
                                                                       save_weights_only=True,
                                                                       verbose=False),
                                           K.callbacks.EarlyStopping(monitor="val_loss",
                                                                     min_delta=0.00001,
                                                                     patience=3,
                                                                     verbose=False)])
            # Load the last saved parameters.
            # The model parameters will correspond to those with the lowest validation loss.
            model.load_weights(best_param_path)

        # Get prediction for inflation in h periods using all data available until the end of the i-th month.
        print("{0} - Predicting...".format(data.index[i]))
        # Get the last lags-many observations of data at point i in time, and normalise it with the normalisation
        # factors computed during the last retraining process.
        prediction_input = data.iloc[i-lags+1:i+1, :] / normalisation_factors
        # Add a new axis so that the input model matches the dimensions (batch_size, timesteps, features).
        prediction_input = prediction_input.to_numpy()[np.newaxis,...]
        # Get the model prediction for prediction_input.
        prediction = model.predict(prediction_input, verbose=False)
        # Get the date of the prediction, i.e., h months from the current i-th point in time.
        prediction_date = data.index[i] + relativedelta(months=h)
        # Save the prediction in the predictions dictionary with the date of the prediction as key.
        # Reshape so that the Numpy array has dimensions (1,). The model returns a Numpy array of dimensions (1,1).
        predictions[prediction_date] = prediction.reshape((1,))

# Create a pandas DataFrame with the predictions and their dates as index.
```

```
predictionsdf = pd.DataFrame.from_dict(predictions, orient="index")

# ------------------------------------------------------------------------------------------------------------------
# Plot prediction results and observed target value.                                                               |
# ------------------------------------------------------------------------------------------------------------------

# Get the plot labels for the x-axis ticks.
xlabels = predictionsdf.index[:-h]
# Get the timeseries of the observations.
observed = all_data.loc[:, "CPIAUCNS"].shift(-h).loc[predictionsdf.index[:-h]]

# Plot the predicted values.
plt.plot(xlabels, predictionsdf.iloc[:-h,0], label="Prediction")
# Plot the observed values.
plt.plot(xlabels, observed, label="Observed")
# Add a legend to the plot in the upper left corner.
plt.legend(loc="upper left")
# Tighten the layout of the plot.
plt.tight_layout()
# Save the resulting plot.
plt.savefig(observed_vs_predicted_plot_path+str(datetime.now()).replace(":", ".")+"_Python_Predicted_vs_Observed_Plot.png")

# ------------------------------------------------------------------------------------------------------------------
# Print predicted vs observed statistical figures.                                                                 |
# ------------------------------------------------------------------------------------------------------------------

# Compute the difference between each prediction and its corresponding observation.
prediction_vs_observed_diffs = predictionsdf.iloc[:-h,0] - observed
# Compute the mean squared error of the predictions w.r.t. the observations.
prediction_vs_observed_mse = np.round(np.mean(np.power(prediction_vs_observed_diffs, 2)), 5)
# Print the mean squared error.
print("Mean squared error (prediction vs observed) {0}.".format(prediction_vs_observed_mse))

# Compute the absolute differences between each prediction and its corresponding observation.
prediction_vs_observed_abs_diffs = np.abs(prediction_vs_observed_diffs)
# Compute the mean absolute error of the predictions w.r.t. the observations.
prediction_vs_observed_mae = np.round(np.mean(prediction_vs_observed_abs_diffs), 5)
# Print the mean absolute error.
print("Mean absolute error (prediction vs observed) {0}.".format(prediction_vs_observed_mae))

# Compute the maximum absolute error of the predictions w.r.t. the observations.
prediction_vs_observed_maxe = np.round(np.max(prediction_vs_observed_abs_diffs), 5)
# Print the maximum absolute error.
print("Maximum absolute error (prediction vs observed) {0}".format(prediction_vs_observed_maxe))
```

C Julia Code

```julia
# This Julia script was written by Francisco Jose Manjon Cabeza Garcia (the author) in the Summer semester
# 2022 for the TU Berlin seminar "Recent Advances in Computational Macroeconomics". The script is a
# practical implementation of a benchmark model, which the author used to compare programming languages for
# researchers. The topic of the seminar paper was based on the paper called "MatLab, Python, Julia: What to
# Choose in Economics?" by Chase Coleman, Spencer Lyon, Lilia Maliar and Serguei Maliar. A presentation of
# this paper was delivered for the seminar, and the paper written by the author shows a small contribution
# to the research direction of the original paper mentioned.
#
# This program is structured in 6 blocks:
# 1. User parameters:
#    Here the user can modify the most important parameters to run the script.
# 2. Import packages and methods:
#    Here the necessary external packages and methods are imported.
# 3. Import and prepare raw data:
#    Here the input data, which also contains the target variable data, is read.
# 4. Define a method to get the neural network model:
#    Here two methods are defined to generate the untrained benchmark model and to train it according to
#    the specifications.
# 5. Create a model and train it at every retraining point:
#    Here the model training and prediction's computations are performed.
# 6. Plot prediction results and observed target value:
#    Here the prediction vs observed plot is generated and prediction accuracy measures are computed.

# ------------------------------------------------------------------------------------------------------------
# User parameters.                                                                                           |
# ------------------------------------------------------------------------------------------------------------

data_csv_path = "C:/Users/labelname/Documents/neural-networks-computational-macroeconomics/Final_data_2022.09.26.csv"
best_param_path = "C:/Users/labelname/Documents/neural-networks-computational-macroeconomics/best_params.bson"
observed_vs_predicted_plot_path = "C:/Users/labelname/Documents/neural-networks-computational-macroeconomics/Plots/"
h = 3                       # Number of months in advance to forecast.
lags = 48                   # Number of past periods to feed into the LSTM cells.
lstm_memory_units = 2       # Number of LSTM cells in the first hidden layer of the network.
dense_layers = 4            # Number of dense layers after the LSTM layer.
dense_layers_units = 128    # Number of cells in each of the hidden dense layers of the network.
val_data_split = 0.25       # Percentage of available data to use for validation at each training point.
warmup_periods = 192        # Number of periods to first train the model.

# ------------------------------------------------------------------------------------------------------------
# Import packages and methods.                                                                               |
# ------------------------------------------------------------------------------------------------------------

using DataFrames            # To handle the input data.
using CSV                   # To read the input data.
using Flux                  # ML package.
using Dates                 # To get attributes of Datetime objects.
using BSON: @save, @load    # To save and load model parameters while training.
using CairoMakie            # Plotting package.

# ------------------------------------------------------------------------------------------------------------
# Import and prepare raw data.                                                                               |
# ------------------------------------------------------------------------------------------------------------

# Import the data we will use to train the model and make predictions.
all_data = CSV.read(data_csv_path,
                    DataFrame,
                    delim=";",
                    header=1)
# Remove the observations from the last h months, because we do not have labels for them yet.
data = all_data[1:end-3, :]

# ------------------------------------------------------------------------------------------------------------
# Define a method to get the neural network model.                                                           |
# ------------------------------------------------------------------------------------------------------------

function Get_Model(;input_size=128, lstm_memory_units=2, dense_layers=4, dense_layers_units=128)
    #=If no parameters are given, this method returns a model as Julia function object with the
    hyperparameters and model specifications of the optimal LSTM-pool model defined in the paper Predicting
    Inflation with Neural Networks by Livia Paranhos.
```

```
    input_size=128          : Number of model features, i.e., data timeseries count
    lstm_memory_units=2     : f_{t|L}-size p
    dense_layers=4          : Layers Q
    dense_layers_units=128  : Nodes n

    LSTM activation function: hyperbolic tangent
    FF-layer activation function: rectified linear unit
    Last FF-layer activation function: linear
    Kernel initialiser: Glorot Uniform
    Bias initialiser: Zeros
    =#

    # For the LSTM layer, init only affects the weights' matrix, not the bias vector.
    lstm_layer = LSTM(input_size => lstm_memory_units,
                      init=Flux.glorot_uniform,
                      initb=Flux.zeros32)
    layers = Vector{Any}([lstm_layer, x -> x[:,end]])
    previous_layer_output_units = lstm_memory_units

    for _ in 1:dense_layers
        # For Dense layers, init only affects the weights' matrix, not the bias vector.
        # The bias vector is initialised to zeros by default.
        push!(layers, Dense(previous_layer_output_units => dense_layers_units,
                            relu,
                            bias=true,
                            init=Flux.glorot_uniform))
        previous_layer_output_units = dense_layers_units
    end

    # The Dense layer activation function is linear by default.
    # For Dense layers, init only affects the weights' matrix, not the bias vector.
    # The bias vector is initialised to zeros by default.
    push!(layers, Dense(previous_layer_output_units => 1,
                        bias=true,
                        init=Flux.glorot_uniform))
    model = Chain(layers)

    return model
end

function train_with_custom_callbacks!(;loss, model, train_data, val_data, opt, epochs, es_patience, es_min_dist)
    #=This method trains a given model with two custom callbacks: one to save the model parameters if the
    validation loss improves, and another one to early stop training if the validation loss does not improve by
    a certain minimum for some consecutive epochs.

    loss            : Flux Loss function with respect to which the model is trained
    model           : Flux model whose parameters will be trained
    train_data      : Vector of tuples (input, label) to use as training data
    val_data        : Vector of tuples (input, label) to use as validation data
    opt             : Flux Optimiser to train the model with
    epochs          : Number of epochs for training
    es_patience     : Maximum epochs after which training is stopped if the validation loss does not improve
    es_min_dist     : Minimum absolute improvement in validation loss

    =#

    # Training_loss is declared local so it will be available for logging outside the gradient calculation.
    local validation_loss

    # Define a variable to track the best validation loss on each epoch. Initialise it to the infinity.
    best_validation_loss = Inf
    # Define a variable to track the number of epochs without validation loss improvement.
    epochs_without_validation_loss_improvement = 0

    # Get the model parameters which are going to be optimised/trained.
    ps = Flux.params(model)

    for _ in 1:epochs
        # Train the model.
        Flux.Optimise.train!(loss, ps, train_data, opt)
```

```
        # Define two vectors to save the validation predictions and labels.
        validation_predictions = Vector{Float32}([])
        validation_labels = Vector{Float32}([])

        for tuple in val_data
            # Compute the prediction for each validation item.
            push!(validation_predictions, model(tuple[1])[1])
            # Save the corresponding label.
            push!(validation_labels, tuple[2])
        end

        # Compute validation loss after training.
        validation_loss = loss(validation_predictions, validation_labels)

        # If the new validation loss is not better than the previous validation loss by at least
        # es_min_dist, count this epoch as one without validation loss improvement.
        if best_validation_loss - validation_loss < es_min_dist
            epochs_without_validation_loss_improvement += 1

        # Else, reset counter for epochs without validation loss improvement, and save the current
        # model parameters.
        else
            @save best_param_path model
            epochs_without_validation_loss_improvement = 0
            best_validation_loss = validation_loss
        end

        # If the number of consecutive epochs without validation loss improvement is greater or equal
        # to the early stopping patience, stop training.
        if epochs_without_validation_loss_improvement >= es_patience
            break
        end
    end
end

# ------------------------------------------------------------------------------------------------------------
# Create a model and train it at every retraining point.                                                     |
# ------------------------------------------------------------------------------------------------------------

# Define a vector to save the model predictions into.
predictions = Vector{Float32}([])
# Define a global model variable.
model = nothing
# Define a global variable for the normalisation factors.
normalisation_factors = nothing

for i in warmup_periods:size(data)[1]
    if Dates.month(data[i,1]) == 12 || model === nothing
        # Either a new year worth of new data is available or this is the start of the loop.
        # Retrain model.
        println(string(data[i,1]) * " - Retraining...")

        # Compute normalisation factors for each feature, so that their values lie in [-1,1].
        # We do not use the data from the last h periods, because we do not have labels for them.
        global normalisation_factors = maximum(abs.(Matrix{Float32}(data[1:i-h, 2:end])'), dims=2)

        # Compute the number of time steps to use as training sample. The rest is used as validation sample.
        training_elements = trunc(Int, (i-h) * (1-val_data_split))
        # Create a training dataset with training_elements-many elements.
        training_dataset = Vector{Any}([])
        # Create a validation dataset with (total-training_elements)-many elements.
        validation_dataset = Vector{Any}([])

        for j in 1:i-lags-h+1
            # A potential source of ambiguity with RNN in Flux can come from the different data layout
            # compared to some common frameworks where data is typically a 3 dimensional array of the form
            # (features, seq length, samples). In Flux, those 3 dimensions are provided through a vector
            # of seq length containing a matrix (features, samples).
            if j < training_elements
                push!(training_dataset, (Matrix{Float32}(data[j:j+lags-1, 2:end])' ./ normalisation_factors,
```

```
                                        data[j+lags+h-1, :CPIAUCNS])) # input data in format (features, lags)
            else
                push!(validation_dataset, (Matrix{Float32}(data[j:j+lags-1, 2:end])' ./ normalisation_factors,
                                           data[j+lags+h-1, :CPIAUCNS])) # input data in format (features, lags)
            end
        end

        # Create a new untrained model.
        global model = Get_Model(input_size=size(data)[2]-1)
        # Train the new model.
        train_with_custom_callbacks!(loss=Flux.Losses.mse,
                                     model=model,
                                     train_data=training_dataset,
                                     val_data=validation_dataset,
                                     opt=Flux.Optimise.Adam(),
                                     epochs=400,
                                     es_patience=3,
                                     es_min_dist=0.00001)

        # Load the last saved paramaters, which showed the lowest validation loss.
        @load best_param_path model
    end

    # Get prediction for inflation in h periods using all data available until the end of the i-th month.
    println(string(data[i,1]) * " - Predicting...")
    # Get the last lags-many observations of data at point i in time, and normalise it with the normalisation
    # factors computed during the last retraining process.
    prediction_input = Matrix{Float32}(data[i-lags+1:i,2:end])' ./ normalisation_factors
    # Get and save the model prediction for prediction_input.
    push!(predictions, model(prediction_input)[1])
end

# ------------------------------------------------------------------------------------------------------------
# Plot prediction results and observed target value.                                                         |
# ------------------------------------------------------------------------------------------------------------

# Get the minimum and maximum year to label the x-axis in the plot.
minyear = Dates.year(data[warmup_periods+h:end, :date][1])
maxyear = Dates.year(data[warmup_periods+h:end, :date][end])

# Create a new Figure object.
fig = Figure(resolution=(640,480))
# Create a new Axis object with year's labels every 24 months (x-axis ticks).
ax = Axis(fig[1,1], xticks=(1:24:length(data[warmup_periods:end-h, :date]), string.(minyear:2:maxyear)))
# Plot the predicted values.
lines!(ax, predictions[1:end-h], label="Prediction")
# Plot the observed values.
lines!(ax, data[warmup_periods:end-h, :CPIAUCNS], label="Observed")
# Add a legend to the plot in the upper left corner.
axislegend(position=:lt)
# Save the resulting plot.
save(observed_vs_predicted_plot_path * "$(replace(string(now()), ":" => "."))_Julia_Predicted_vs_Observed_Plot.png", fig)

# ------------------------------------------------------------------------------------------------------------
# Print predicted vs observed statistical figures.                                                           |
# ------------------------------------------------------------------------------------------------------------

# Compute the difference between each prediction and its corresponding observation.
prediction_vs_observed_diffs = predictions[1:end-h] - Vector{Float32}(data[warmup_periods:end-h, :CPIAUCNS])
# Compute the mean squared error of the predictions w.r.t. the observations.
prediction_vs_observed_mse = round((1/length(predictions[1:end-h]))*sum(prediction_vs_observed_diffs.^2),
                                   digits=5)
# Print the mean squared error.
println("Mean squared error (prediction vs observed) $prediction_vs_observed_mse.")

# Compute the absolute differences between each prediction and its corresponding observation.
prediction_vs_observed_abs_diffs = abs.(prediction_vs_observed_diffs)
# Compute the mean absolute error of the predictions w.r.t. the observations.
prediction_vs_observed_mae = round((1/length(predictions[1:end-h]))*sum(prediction_vs_observed_abs_diffs),
                                   digits=5)
# Print the mean absolute error.
```

```
println("Mean absolute error (prediction vs observed) $prediction_vs_observed_mae.")

# Compute the maximum absolute error of the predictions w.r.t. the observations.
prediction_vs_observed_maxe = round(maximum(prediction_vs_observed_abs_diffs), digits=5)
# Print the maximum absolute error.
println("Maximum absolute error (prediction vs observed) $prediction_vs_observed_maxe.")
```