Technical Report: Implementing a Fully Convolutional Neural Network with Long Short-Term Memory for the `sits` R Library

Alexandre Assunção
alexcarssuncao@gmail.com

Advisor: Clodoveu Davis
clodoveu@dcc.ufmg.br

Co-advisor: Gilberto Câmara
gilberto.camara.inpe@gmail.com

January 28, 2025

Abstract

The integration of machine and deep learning models into Earth science applications has become increasingly crucial for analyzing time-series satellite data, particularly in monitoring environmental changes and land cover classification. This report presents the development and implementation of the `sits_lstm_fcn` function within the SITS package, an open-source R framework for satellite image time-series analysis. Based on the LSTM Fully Convolutional Networks (LSTM-FCN) model, this function combines temporal convolutions and LSTM blocks to effectively capture both local temporal patterns and long-term dependencies in the data. The report details the theoretical background, model architecture, and training methodology, along with the challenges and solutions encountered during implementation. Results from experiments on multi-band satellite data demonstrate the model's strong classification performance, achieving 85% accuracy on the validation dataset. By providing a user-friendly interface, this contribution expands the SITS package's capabilities, enabling researchers to utilize advanced deep learning models with minimal programming effort.

Introduction

Monitoring Earth's changing environment is crucial for understanding and mitigating the impacts of climate change. Satellite imagery plays a key role in this effort, providing rich temporal data for analyzing land use, vegetation dynamics, and other phenomena. However, transforming this data into actionable insights often requires expertise in advanced techniques like machine and deep learning—skills many Earth scientists lack.

To address this challenge, tools like SITS (an open-source R package) have been developed to simplify the analysis of time-series satellite data. The SITS API allows researchers to use a wide range of machine learning and deep learning algorithms with minimal programming or mathematical expertise.

Given the fast-paced evolution of machine learning, new methods are continually being developed and integrated. In this project, I contributed to the SITS API by implementing a deep learning method: a fully convolutional network (FCN) with temporal convolutions and long short-term memory (LSTM) blocks. SITS users are able to train this model and classify time-series satellite imagery data with only a few lines of R code, making sophisticated analyses more accessible to a wider scientific community.

Theoretical Background

The new API function implemented in SITS was proposed in the *LSTM Fully Convolutional Networks for Time Series Classification*, by Fazle Karim et al., published by *IEEE Access*, 6(1662-1669), in 2018. It is a branched neural network with two paths running concurrently: dimension shuffling and LSTM, and an FCN pathway.

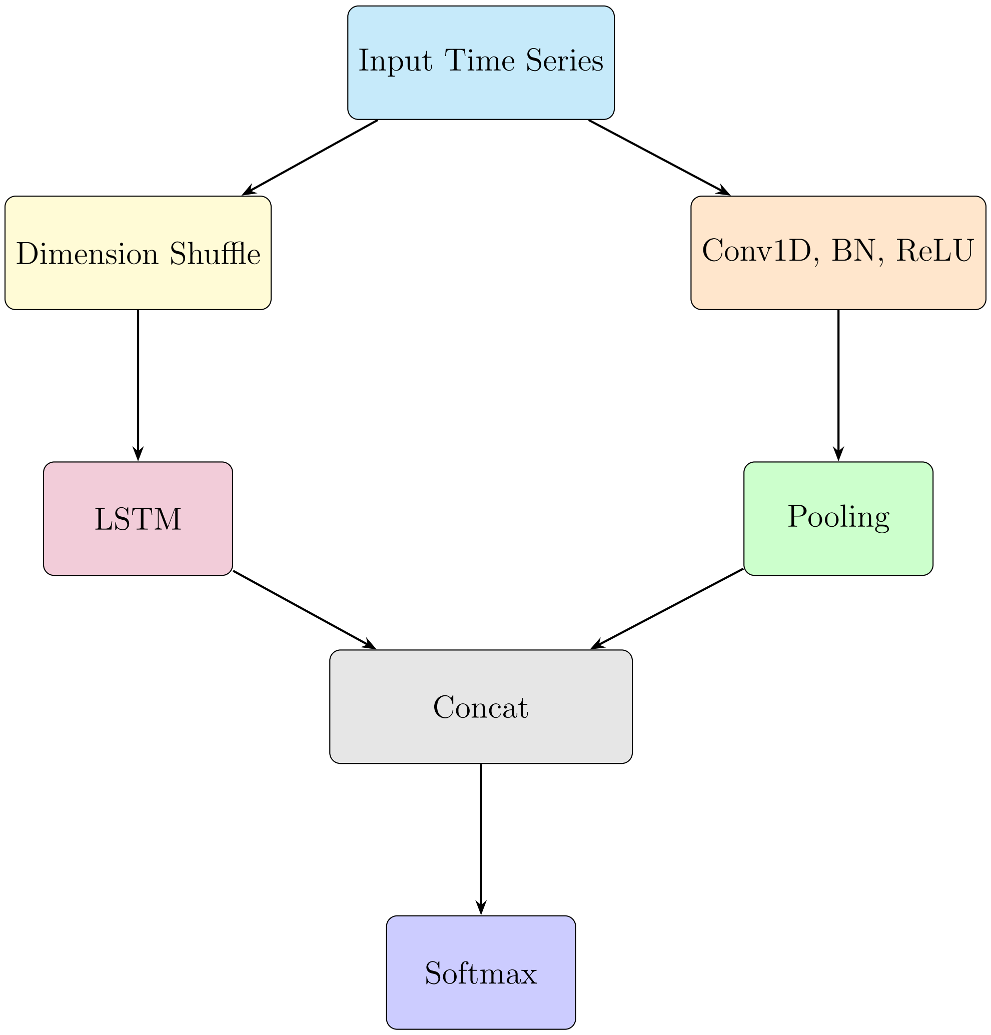


Figure 1: Network Architecture Diagram

LSTM Pathway

For clarity, I assume each time series has a single band—Normalized Difference Vegetation Index (NDVI)—though this discussion can be easily expanded to include multivariable time series as well. When each predictor enters this pathway, the dimension shuffling layer makes it so that the LSTM sees each time step in the series (e.g., each date's NDVI value) as a separate feature or *variable* within a single time step. Instead of observing a sequence of time steps, it sees a single *frame* containing all those time steps as different variables.

With this reshaped input, the LSTM layer then attempts to capture dependencies among the transformed time steps within that single series. It focuses on relationships within each individual time series after it's been reshaped.

FCN Pathway

The FCN pathway applies 1D temporal convolutions to a time series by sliding a filter across the time steps to capture local temporal patterns within and across each variable. This process is analogous to how convolutions are applied to images. Just as an image has bands (e.g., red, green, and blue), a time series has variables—such as NDVI, Enhanced Vegetation Index (EVI), and Near Infrared (NIR)—which can each be filtered to extract relevant patterns. Following the convolutional layers, global pooling reduces the dimensionality, condensing information across bands.

Both pathways run in parallel and are merged before applying softmax for classification.

Implementation Details

Preprocessing

- Input samples are normalized using `.pred_normalize()` based on the statistics of the training data.
- If no validation dataset is provided, the training dataset is split into training and validation sets.

Model Architecture

LSTM Branch

- Processes the time series data using an LSTM layer.
- Includes a dropout layer for regularization.

FCN Branch

- Applies three 1D convolutional layers with batch normalization and ReLU activation.
- Features dropout for each convolutional layer.
- Includes global average pooling to reduce spatial dimensions.

Dense Layer

- Concatenates the outputs of the LSTM and FCN branches.
- Applies a fully connected layer for classification.

Training

The model is trained using the `luz` package with:

- Cross-entropy loss.
- Accuracy as a performance metric.
- Callbacks for early stopping and learning rate scheduling.

Function Signature

```
sits_lstm_fcn(
  samples = NULL,
  samples_validation = NULL,
  cnn_layers = c(128, 256, 128),
  cnn_kernels = c(8, 5, 3),
  cnn_dropout_rates = c(0.0, 0.0, 0.0),
  lstm_width = 8,
  lstm_dropout = 0.8,
  epochs = 50,
  batch_size = 64,
  validation_split = 0.2,
  optimizer = torch::optim_adamw,
  opt_hparams = list(lr = 5.0e-04, eps = 1.0e-08, weight_decay = 1.0e-06),
  lr_decay_epochs = 1,
  lr_decay_rate = 0.95,
  patience = 20,
  min_delta = 0.01,
  verbose = FALSE
)
```

Results

Dataset Description

The classification experiments were conducted on the Sinop and Rondonia raster cubes available in the sits package. The samples used in the training (available in the sitsdata package) contain time-series satellite data, with key features including NDVI, Enhanced Vegetation Index (EVI) values, for the Sinop cube, and NDVI, EVI, Near-Infrared (NIR), and Mid-Infrared MIR values for the Rondonia cube. The data spans multiple seasons and captures various land cover types, providing a robust test of the model's ability to distinguish between classes.

Experimental Setup

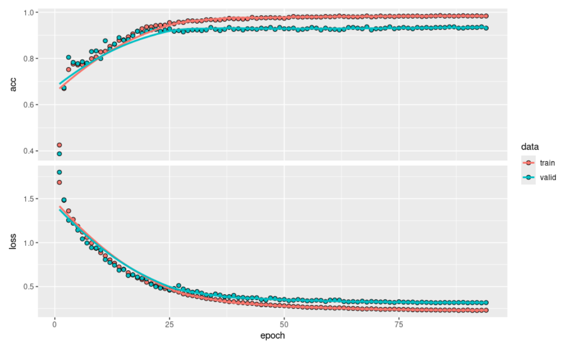
Following the extensive tuning the model's authors performed on their paper, the `sits_lstm_fcn` function was configured with the following parameters, which were found to be optimal:

- Convolutional layers: `[128, 256, 128]`.
- Kernel sizes: `[8, 5, 3]`.
- LSTM width: `8`.
- Epochs: `50`.
- Batch size: `64`.

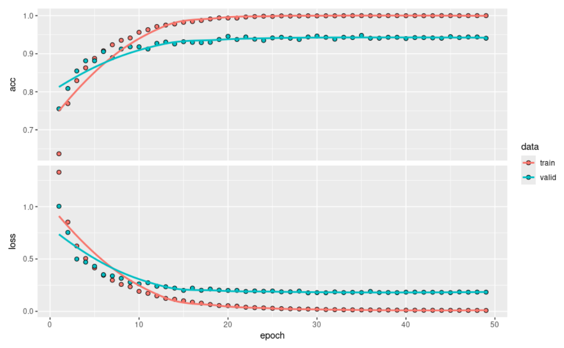
The training and validation datasets were split with an 80-20 ratio, and the model was trained using the AdamW optimizer with a learning rate of `5e-4`.

Performance Metrics

Using the `plot(sits_lstm_fcn_model)` function, we can see the model's loss and accuracy for both the training and validating sets.



(a) Training with the Mato Grosso samples.



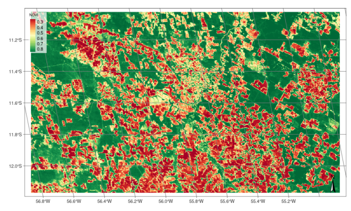
(b) Training with the Rondonia samples.

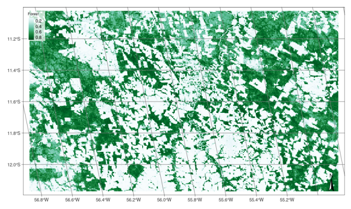
Figure 2: Training the model.

As we can see, the model quickly converges, resulting in early stops for both data. However, it is worth pointing out that the datasets included in the sitsdata package are very well-behaved are easy to classify.

Visualization

Below are two examples of the classification maps generated from the model's predictions:


(a) Sinop NDVI


(b) Sinop Forest Probabilities

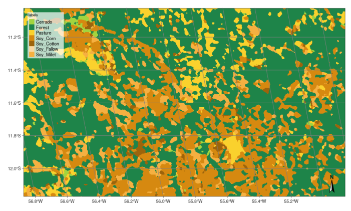
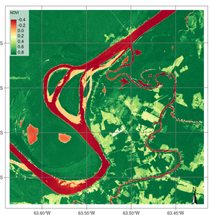
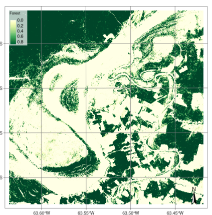

(c) Sinop Classified

Figure 3: Sinop Raster Cube Classification.


(a) Rondonia NDVI


(b) Rondonia Forest Probabilities

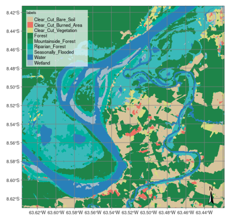

(c) Rondonia Classified

Figure 4: Rondonia Raster Cube Classification.

Student's Contribution

- Learning the Model: The project began with an in-depth study of the paper "LSTM Fully Convolutional Networks for Time Series Classification" to understand the model's architecture and principles. This was essential for translating the theory into practice.
- Familiarity with R and Libraries: As part of this project, I became proficient in R, specifically its implementation of PyTorch (‘torch‘), and the use of supporting libraries like ‘luz‘ and ‘tidyverse‘. These tools were fundamental in building and managing the model.
- Reimplementation of SITS Functions: To facilitate the implementation, I created custom versions of many internal SITS functions, particularly those responsible for data formatting and parameter testing. This effort helped isolate and debug the components necessary for integrating the model.
- Visualization Functions: The visualization functions from SITS were not reimplemented, as I am still learning their use and functionality. This area remains a focus for future development.
- Transition to the SITS Package: Once the model worked in a microenvironment, I integrated it into the SITS package, leveraging internal functions. This phase involved resolving compatibility issues related to operating systems and CUDA support, ensuring seamless execution across platforms.

References

[1] F. Karim, S. Majumdar, H. Darabi, and S. Chen, "LSTM Fully Convolutional Networks for Time Series Classification," *IEEE Access*, vol. 6, pp. 1662–1669, 2018. doi: 10.1109/ACCESS.2017.2779939.

[2] J. Long, E. Shelhamer, and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," *ArXiv*, 2014. Available: https://arxiv.org/abs/1411.4038.

[3] R. Simoes, G. Camara, G. Queiroz, F. Souza, P. R. Andrade, L. Santos, A. Carvalho, and K. Ferreira, "Satellite Image Time Series Analysis for Big Earth Observation Data," *Remote Sensing*, vol. 13, p. 2428, 2021.

A Function Source Code

The source code for the `sits_lstm_fcn` function is provided below:

```
#' @title Train an lstm and fcn neural network
#' @name sits_lstm_fcn
#'
#' @author Alexandre Assuncao, \email{alexcarssuncao@@gmail.com}
#'
#' @description Uses a branched neural network consisting of
#'  a lstm branch and a three-layer fully convolutional branch
#'  followed by concatenation to classify time series data.
#'
#' This function is based on the paper by Fazle Karim, Somshubra Majumdar,
#' and Houshang Darabi. If you use this method, please cite the original
#' LSTM with FCN paper.
#'
#' The torch version is based on the code made available by the titu1994.
#' The original python code is available at the website
#' https://github.com/titu1994/LSTM-FCN. This code is licensed as GPL-3.
#'
#' @references F. Karim, S. Majumdar, H. Darabi and S. Chen,
#' "LSTM Fully Convolutional Networks for Time Series Classification,"
#'  in IEEE Access, vol. 6, pp. 1662-1669, 2018,
#'  doi: 10.1109/ACCESS.2017.2779939.
#'
#' @param samples            Time series with the training samples.
#' @param samples_validation Time series with the validation samples. if the
#'                           \code{samples_validation} parameter is provided,
#'                           the \code{validation_split} parameter is ignored.
#' @param lstm_width         Number of neuros in the lstm's hidden layer.
#' @param lstm_dropout       Dropout rate of the lstm layer.
#' @param cnn_layers         Number of 1D convolutional filters per layer
#' @param cnn_kernels        Size of the 1D convolutional kernels.
#' @param cnn_dropout_rates  Dropout rates for 1D convolutional filters.
#' @param epochs             Number of iterations to train the model.
#' @param batch_size         Number of samples per gradient update.
#' @param validation_split   Fraction of training data to be used for
#'                           validation.
#' @param optimizer          Optimizer function to be used.
#' @param opt_hparams        Hyperparameters for optimizer:
#'                           lr : Learning rate of the optimizer
#'                           eps: Term added to the denominator
#'                                to improve numerical stability.
#'                           weight_decay:       L2 regularization
#' @param lr_decay_epochs    Number of epochs to reduce learning rate.
#' @param lr_decay_rate      Decay factor for reducing learning rate.
#' @param patience           Number of epochs without improvements until
```

```r
#'                             training stops.
#' @param min_delta          Minimum improvement in loss function
#'                             to reset the patience counter.
#' @param verbose            Verbosity mode (TRUE/FALSE). Default is FALSE.
#'
#' @return A fitted model to be used for classification.
#'
#' @export

sits_lstm_fcn <- function(samples = NULL,
                          samples_validation = NULL,
                          cnn_layers = c(128, 256, 128),
                          cnn_kernels = c(8, 5, 3),
                          cnn_dropout_rates = c(0.0, 0.0, 0.0),
                          lstm_width = 8,
                          lstm_dropout = 0.8,
                          epochs = 50,
                          batch_size = 64,
                          validation_split = 0.2,
                          optimizer = torch::optim_adamw,
                          opt_hparams = list(
                              lr = 5.0e-04,
                              eps = 1.0e-08,
                              weight_decay = 1.0e-06
                          ),
                          lr_decay_epochs = 1,
                          lr_decay_rate = 0.95,
                          patience = 20,
                          min_delta = 0.01,
                          verbose = FALSE) {
    # set caller for error msg
    .check_set_caller("sits_lstm_fcn")
    # Function that trains a torch model based on samples
    train_fun <- function(samples) {
        # does not support working with DEM or other base data
        if (inherits(samples, "sits_base"))
            stop(.conf("messages", "sits_train_base_data"), call. = FALSE)
        # Avoid add a global variable for 'self'
        self <- NULL
        # Verifies if 'torch' and 'luz' packages is installed
        .check_require_packages(c("torch", "luz"))
        # Pre-conditions:
        .check_samples_train(samples)
        .check_int_parameter(cnn_layers, len_max = 2^31 - 1)
        .check_int_parameter(cnn_kernels,
                             len_min = length(cnn_layers),
                             len_max = length(cnn_layers)
        )
```

```
.check_num_parameter(cnn_dropout_rates, min = 0, max = 1,
                     len_min = length(cnn_layers), len_max = length(cnn_layer
)
.check_int_parameter(lstm_width, len_max = 2^31 - 1)
.check_num_parameter(lstm_dropout, min = 0, max = 1)
.check_int_parameter(epochs)
.check_int_parameter(batch_size)
# Check validation_split parameter if samples_validation is not passed
if (is.null(samples_validation)) {
    .check_num_parameter(validation_split, exclusive_min = 0, max = 0.5)
}
# Check opt_hparams
# Get parameters list and remove the 'param' parameter
optim_params_function <- formals(optimizer)[-1]
if (!is.null(opt_hparams)) {
    .check_lst_parameter(opt_hparams,
                         msg = .conf("messages", ".check_opt_hparams")
    )
    .check_chr_within(
        x = names(opt_hparams),
        within = names(optim_params_function),
        msg = .conf("messages", ".check_opt_hparams")
    )
    optim_params_function <- utils::modifyList(
        x = optim_params_function, val = opt_hparams
    )
}
# Other pre-conditions:
.check_int_parameter(lr_decay_epochs)
.check_num_parameter(lr_decay_rate, exclusive_min = 0, max = 1)
.check_int_parameter(patience)
.check_num_parameter(min_delta, min = 0)
.check_lgl_parameter(verbose)
# Samples labels
labels <- .samples_labels(samples)
# Samples bands
bands <- .samples_bands(samples)
# Samples timeline
timeline <- .samples_timeline(samples)
# Create numeric labels vector
code_labels <- seq_along(labels)
names(code_labels) <- labels
# Number of labels, bands, and number of samples (used below)
n_labels <- length(labels)
n_bands <- length(bands)
n_times <- .samples_ntimes(samples)
# Data normalization
ml_stats <- .samples_stats(samples)
```

```r
train_samples <- .predictors(samples)
train_samples <- .pred_normalize(pred = train_samples, stats = ml_stats)
# Post condition: is predictor data valid?
.check_predictors(pred = train_samples, samples = samples)
# Are there validation samples?
if (!is.null(samples_validation)) {
    .check_samples_validation(
        samples_validation = samples_validation, labels = labels,
        timeline = timeline, bands = bands
    )
    # Test samples are extracted from validation data
    test_samples <- .predictors(samples_validation)
    test_samples <- .pred_normalize(
        pred = test_samples, stats = ml_stats
    )
} else {
    # Split the data into training and validation data sets
    # Create partitions different splits of the input data
    test_samples <- .pred_sample(
        pred = train_samples, frac = validation_split
    )
    # Remove the lines used for validation
    sel <- !train_samples[["sample_id"]] %in%
        test_samples[["sample_id"]]
    train_samples <- train_samples[sel, ]
}
n_samples_train <- nrow(train_samples)
n_samples_test <- nrow(test_samples)
# Shuffle the data
train_samples <- train_samples[sample(
    nrow(train_samples), nrow(train_samples)
), ]
test_samples <- test_samples[sample(
    nrow(test_samples), nrow(test_samples)
), ]
# Organize data for model training
train_x <- array(
    data = as.matrix(.pred_features(train_samples)),
    dim = c(n_samples_train, n_times, n_bands)
)
train_y <- unname(code_labels[.pred_references(train_samples)])
# Create the test data
test_x <- array(
    data = as.matrix(.pred_features(test_samples)),
    dim = c(n_samples_test, n_times, n_bands)
)
test_y <- unname(code_labels[.pred_references(test_samples)])
# Set torch seed
```

```
torch::torch_manual_seed(sample.int(10^5, 1))
# The LSTM/FCN for time series:
lstm_fcn_model <- torch::nn_module(
    classname = "model_lstm_fcn",
    initialize = function(n_bands,
                          n_times,
                          n_labels,
                          kernel_sizes,
                          hidden_dims,
                          lstm_width,
                          cnn_dropout_rates,
                          lstm_dropout) {
        # Upper branch: LSTM with dimension shift
        self$lstm <- torch::nn_lstm(
            input_size = n_times,
            hidden_size = lstm_width,
            dropout = 0,
            num_layers = 1,
            batch_first = TRUE
        )
        # Lstm's dropout
        self$dropout <- torch::nn_dropout(p = lstm_dropout)
        # Lower branch: Fully Convolutional Layers and avg pooling
        self$conv_bn_relu1 <- .torch_conv1D_batch_norm_relu_dropout(
            input_dim = n_bands,
            output_dim = hidden_dims[[1]],
            kernel_size = kernel_sizes[[1]],
            padding = as.integer(kernel_sizes[[1]] %/% 2),
            dropout_rate = cnn_dropout_rates[[1]]
        )
        self$conv_bn_relu2 <- .torch_conv1D_batch_norm_relu_dropout(
            input_dim = hidden_dims[[1]],
            output_dim = hidden_dims[[2]],
            kernel_size = kernel_sizes[[2]],
            padding = as.integer(kernel_sizes[[2]] %/% 2),
            dropout_rate = cnn_dropout_rates[[2]]
        )
        self$conv_bn_relu3 <- .torch_conv1D_batch_norm_relu_dropout(
            input_dim = hidden_dims[[2]],
            output_dim = n_bands,
            kernel_size = kernel_sizes[[3]],
            padding = as.integer(kernel_sizes[[3]] %/% 2),
            dropout_rate = cnn_dropout_rates[[3]]
        )
        # Global average pooling
        self$pooling <- torch::nn_adaptive_avg_pool1d(output_size = lstm_widt
        # Flattening 3D tensor to run the dense layer
        self$flatten <- torch::nn_flatten()
```

```
            # Final module: dense layer outputting the number of labels
            self$dense <- torch::nn_linear(
                in_features = n_bands * lstm_width * 2,
                out_features = n_labels
            )
        },
        forward = function(x) {
            # dimension shift and LSTM forward pass
            x_lstm <- x$permute(c(1, 3, 2)) |>
                self$lstm()
            # FCN forward pass
            x_fcn <- x$permute(c(1, 3, 2)) |>
                self$conv_bn_relu1() |>
                self$conv_bn_relu2() |>
                self$conv_bn_relu3() |>
                self$pooling()
            # Concatenate upper and lower branches
            x_combined <- torch::torch_cat(list(x_lstm[[1]], x_fcn), dim = 2)
            x_flat <- self$flatten(x_combined)
            x_out <- x_flat |>
                self$dense()
        }
    )
    # train with CPU or GPU?
    cpu_train <- .torch_cpu_train()
    # Train the model using luz
    torch_model <-
        luz::setup(
            module = lstm_fcn_model,
            loss = torch::nn_cross_entropy_loss(),
            metrics = list(luz::luz_metric_accuracy()),
            optimizer = optimizer
        ) |>
        luz::set_opt_hparams(
            !!!optim_params_function
        ) |>
        luz::set_hparams(
            n_bands = n_bands,
            n_times = n_times,
            n_labels = length(labels),
            kernel_sizes = cnn_kernels,
            hidden_dims = cnn_layers,
            lstm_width = lstm_width,
            cnn_dropout_rates = cnn_dropout_rates,
            lstm_dropout = lstm_dropout
        ) |>
        luz::fit(
            data = list(train_x, train_y),
            epochs = epochs,
            valid_data = list(test_x, test_y),
            callbacks = list(
                luz::luz_callback_early_stopping(
                    monitor = "valid_loss",
                    patience = patience,
                    min_delta = min_delta,
                    mode = "min"
                ),
                luz::luz_callback_lr_scheduler(
                    torch::lr_step,
                    step_size = lr_decay_epochs,
                    gamma = lr_decay_rate
                )
            ),
            accelerator = luz::accelerator(cpu = TRUE),
            dataloader_options = list(batch_size = batch_size),
            verbose = verbose
        )
    # Serialize model
    serialized_model <- .torch_serialize_model(torch_model[["model"]])

    # Function that predicts labels of input values
    predict_fun <- function(values) {
        # Verifies if torch package is installed
        .check_require_packages("torch")
        # Set torch threads to 1
        # Note: function does not work on MacOS
        suppressWarnings(torch::torch_set_num_threads(1))
        # Unserialize model
        torch_model[["model"]] <- .torch_unserialize_model(serialized_model)
        # Used to check values (below)
        input_pixels <- nrow(values)
        # Transform input into a 3D tensor
        # Reshape the 2D matrix into a 3D array
        n_samples <- nrow(values)
        n_times <- .samples_ntimes(samples)
        n_bands <- length(bands)
        # Performs data normalization
        values <- .pred_normalize(pred = values, stats = ml_stats)
        # Represent matrix values as array
        values <- array(
            data = as.matrix(values), dim = c(n_samples, n_times, n_bands)
        )
        # Get GPU memory
        gpu_memory <- sits_env[["gpu_memory"]]
        # if CUDA is available and gpu memory is defined, transform values
        # to torch dataloader
```

```
            if (.torch_has_cuda() && .has(gpu_memory)) {
                # set the batch size according to the GPU memory
                b_size <- 2^gpu_memory
                # transfor the input array to a dataset
                values <- .as_dataset(values)
                # To the data set to a torch transform in a dataloader to use the bat
                values <- torch::dataloader(values, batch_size = b_size)
                # Do GPU classification with dataloader
                values <- .try(
                    stats::predict(object = torch_model, values),
                    .msg_error = .conf("messages", ".check_gpu_memory_size")
                )
            } else {
                # Do  classification without dataloader
                values <- stats::predict(object = torch_model, values)
            }
            # Convert from tensor to array
            values <- torch::as_array(values)
            # Update the columns names to labels
            colnames(values) <- labels
            return(values)
        }
        # Set model class
        predict_fun <- .set_class(
            predict_fun, "torch_model", "sits_model", class(predict_fun)
        )
        return(predict_fun)
    }
    # If samples is informed, train a model and return a predict function
    # Otherwise give back a train function to train model further
    result <- .factory_function(samples, train_fun)
    return(result)
}
```

B Link to SITS GitHub Repository

The full SITS package, including the `sits_lstm_fcn` function, is available at the following GitHub repository: https://github.com/e-sensing/sits (In the dev branch).

Declaration of AI Assistance

Generative AI tools were used in the preparation of this document to improve language clarity, structure, and style. The content and technical aspects remain my original work.