mltype

Jan Krepl

Nov 10, 2020

CONTENTS:

`mltype` is a terminal application for improving typing speed and accuracy. It does so with a tiny bit of deep learning.

CHAPTER

ONE

INSTALLATION

The simplest way to install `mltype` is via PyPI

```
pip install mltype
```

To get the latest version or potentially help with developlment, clone the github repository

```
git clone https://github.com/jankrepl/mltype.git
cd mltype
pip install -e .
```

1.1 Extra dependencies

One can use the following sytax to install extra dependencies

```
pip install -e .[GROUP]
```

Below are the available groups with

- `dev` - development tools
- `hecate` - tools for running optional curses tests
- `mlflow` - optional tracking tool to visualize training progress

CHAPTER

TWO

COMMAND LINE INTERFACE

The command line interface (CLI) is the primary way of using `mltype`. After installation, one can use the entrypoint `mlt` that is going to be in the path.

```
$ mlt
Usage: mlt [OPTIONS] COMMAND [ARGS]...

  Tool for improving typing speed and accuracy

Options:
  --help  Show this message and exit.

Commands:
  file    Type text from a file.
  ls      List all language models
  random  Sample characters randomly from a provided vocabulary
  raw     Provide text manually
  replay  Compete against a past performance
  sample  Sample text from a language
  train   Train a language
```

Note that `mltype` uses the folder `~/.mltype` (in the home directory) for storing all relevant data. See below the usual structure.

```
- .mltype/
   - checkpoints/
       - a/  # training checkpoints of model a
       - b/  # training checkpoints of model b
   - languages/
       - a  # some model
       - b  # some other model
       ...
   - logs/
      ..
```

2.1 file

Type random (or fixed) lines from a text file. This command has two main modes:

1. **Random lines** - Select random consecutive lines. One needs to specify `--n-lines` and optionally the `random-state` (for reproducibility).
2. **Fixed lines** - One needs to specify `--start-line` and `--end-line`.

2.1.1 Arguments

- `PATH` - Path to the text file to read from

2.1.2 Options

- `-e, --end-line INTEGER` - The end line of the excerpt to use. Needs to be used together with start-line.
- `-f, --force-perfect` - All characters need to be typed correctly
- `-i, --instant-death` - End game after the first mistake
- `-l, --n-lines INTEGER` - Number of consecutive lines to be selected at random. Cannot be used together with start-line and end-line.
- `-o, --output-file PATH` - Path to where to save the result file
- `-r, --random-state INTEGER`
- `-s, --start-line INTEGER` - the start line of the excerpt to use. needs to be used together with end-line.
- `-t, --target-wpm INTEGER` - The desired speed to be shown as a guide
- `-w, --include-whitespace` - Include whitespace characters.

2.1.3 Examples

Let us first create a text file

```
echo $'zeroth\nfirst\nsecond\nthird\nfourth\nfifth\nsixth' > text.txt
cat text.txt
```

```
zeroth
first
second
third
fourth
fifth
sixth
```

To select contiguous lines randomly, one can to specify `-l, --n_lines` representing the number of lines to use.

```
mlt file -l 2 text.txt
```

Which would open the typing interface with 2 random contiguous lines

```
second third
```

The other option would be to use the deterministic mode and select the starting and ending line manually

```
mlt file -s 0 -e 3 text.txt
```

```
zeroth first second
```

As multiple commands, one can specify a target speed and an output file. Note that we follow the Python convention - line counting starts from zero and the intervals contain the starting line but not the ending one.

Note that one can keep the whitespace characters (including newlines) in the text by adding the `-w, --include_whitespace` option

```
mlt file -l 2 -w text.txt
```

```
second
third
```

2.2 ls

List available language models. One can use them with *sample*.

Please check the official github to download pretrained models - mltype github.

Note: `mlt ls` simply lists all the files present in `~.mltype/languages`.

2.2.1 Examples

```
mlt ls
```

```
python
some_amazing_model
wikipedia
```

2.3 random

Generate random sequence of characters based on provided counts. The absolute counts are converted to relative counts (probability distribution) that we sample from.

Note: `mlt random` samples characters independently unlike `mlt sample` which conditions on previous characters.

2.3.1 Arguments

- `CHARACTERS` - Characters to include in the vocabulary. The higher the number of occurances of a given character the higher the probabilty of this character being sampled.

2.3.2 Options

- `-f, --force-perfect` - All characters need to be typed correctly
- `-i, --instant-death` - End game after the first mistake
- `-n, --n-chars INTEGER` - Number of characters to sample
- `-o, --output-file PATH` - Path to where to save the result file
- `-t, --target-wpm INTEGER` - The desired speed to be shown as a guide

2.3.3 Examples

Let's say we want to practise typing of digits. However, we would like to spend more time on 5's and 6's since they are harder.

```
mlt random "123455556666789    "
```

This would give us something like this.

```
546261561 3566  53 5496 556659554 435 1386559569  5 85641553465118589
```

We see that the most frequent characters are 5's, 6's and spaces.

2.4 raw

Provide text manually.

2.4.1 Arguments

- `TEXT` - Text to be transfered to the typing interface

2.4.2 Options

- `-f, --force-perfect` - All characters need to be typed correctly
- `-i, --instant-death` - End game after the first mistake
- `-o, --output-file PATH` - Path to where to save the result file
- `-r, --raw-string` - If active, then newlines and tabs are not seen as special characters
- `-t, --target-wpm INTEGER` - The desired speed to be shown as a guide

2.4.3 Examples

Let's say we have some text in the clipboard that we just paste and type. Additionally, we want to see the 80 word per minute (WPM) marker. Lastly, no errors are acceptable—instant death mode.

```
mlt raw -i -t 80 "Hello world I will write you quickly"
```

```
Hello world I will write you quickly
```

2.5 replay

Play against a past performance. To save a past performance one can use the option `-o, --output_file` of the following commands

- *file*
- *random*
- *raw*
- *sample*

2.5.1 Arguments

- `REPLAY_FILE` - Past performance to play against

2.5.2 Options

- `-f, --force-perfect` - All characters need to be typed correctly
- `-i, --instant-death` - End game after the first mistake
- `-t, --target-wpm INTEGER` - The desired speed to be shown as a guide
- `-w, --overwrite PATH` - Overwrite in place if faster

2.5.3 Examples

We ran `mlt sample ... -o replay_file` and we are not particularly happy about the performance. We would like to replay the same text and try to improve our speed. In case we do, we would like the `replay_file` to be updated automatically (using the `-w, --overwrite` option).

```
mlt replay -w replay_file
```

```
Some text we already typed before.
```

2.6 sample

Generate text using a character-level language model.

Note: As opposed to `mlt random`, the `mlt sample` command is taking into consideration all the previous characters and therefore could generate more realistic text.

To see all the available models use *ls*. Please check the official github to download pretrained models - mltype github.

2.6.1 Arguments

- `MODEL_NAME` - Name of the language model

2.6.2 Options

- `-f, --force-perfect` - All characters need to be typed correctly
- `-i, --instant-death` - End game after the first mistake
- `-k, --top-k INTEGER` - Consider only the top k most probable characters
- `-n, --n-chars INTEGER` - Number of characters to generate
- `-o, --output-file PATH` - Path to where to save the result file
- `-r, --random-state INTEGER` - Random state for reproducible results
- `-s, --starting-text TEXT` - Initial text used as a starting condition
- `-t, --target-wpm INTEGER` - The desired speed to be shown as a guide
- `-v, --verbose` Show progressbar when generating text

2.6.3 Examples

We want to practise typing Python without having to worry about having real source code. Assuming we have a decent language model for Python (see *train*) called `amazing_python_model` then we can do the following

```
mlt sample amazing_python_model
```

```
spatial_median(X, method="lar", call='Log', Cov']) glm.fit(X, y) assert_all
close(ref_no_encoded_c
```

Maybe we would like to give the model some initial text and let it complete it for us.

```
mlt sample -s "@pytest.mark.parametrize" amazing_python_model
```

```
@pytest.mark.parametrize('solver', ['sparse_cg', 'sag', 'saga'])
@pytest.mark.parametrize('copy_X', ['not a number', -0.10]]
```

2.7 train

Train a character-level language model. The trained model can then be used with *sample*.

In the background, we use an LSTM and feedforward network architecture to achieve this task. The user can set most of the important hyperparameters via the CLI options. Note that one can train without a GPU, however, to get access to bigger networks and faster training (~minutes/hours) GPUs are recommended.

2.7.1 Arguments

- `PATH_1, PATH_2, ...` - Paths to files or folders containing text to be trained on
- `MODEL_NAME` - Name of the trained model

2.7.2 Options

- `-b, --batch-size INTEGER` - Number of samples in a batch
- `-c, --checkpoint-path PATH` - Load a checkpoiont and continue training it
- `-d, --dense-size INTEGER` - Size of the dense layer
- `-e, --extensions TEXT` - Comma-separated list of allowed extensions
- `-f, --fill-strategy TEXT` - Either zeros or skip. Determines how to deal with out of vocabulary characters
- `-g, --gpus INTEGER` - Number of gpus. In not specified, then none. If -1, then all.
- `-h, --hidden_size INTEGER` - Size of the hidden state
- `-i, --illegal-chars TEXT` - Characters to exclude from the vocabulary.
- `-l, --n-layers` :code`INTEGER` - Number of layesr in the recurrent network
- `-m, --use-mlflow` - Use MLFlow for logging
- `-n, --max-epochs INTEGER` - Maximum number of epochs
- `-o, --output-path PATH` - Custom path where to save the trained models and logging details. If not provided it defaults to *~/.mltype*.
- `-s, --early-stopping` - Enable early stopping based on validation loss
- `-t, --train-test-split FLOAT` - Train test split - value between (0, 1)
- `-v, --vocab-size INTEGER` - Number of the most frequent characters to include in the vocabulary
- `-w, --window-size INTEGER` - Number of previous characters to consider for prediction

2.7.3 Examples

Let's assume we have a book in fulltext saved in the `book.txt` file. Our goal would be to train a model that learns the language used in this book and is able to sample new pieces of text that resemble the original.

See below a list of hyperparameters that work reasonably well and the training can be done in a few hours (on a GPU)

- `--batch-size` 128
- `--dense-size` 1024
- `--early-stopping`
- `--gpus` 1
- `--hidden-size` 512
- `--max-epochs` 10
- `--n-layers` 3
- `--vocab-size` 70
- `--window-size` 100

So overall the commands looks like

```
mlt train book.txt cool_model -n 3 -s -g 1 -b 128 -l 3 -h 512 -d 1024 -w 100 -v 80
```

During the training, one can see progress bars and the training and validation loss (using `pytorch-lightning` in the background). Once the training is done, the best model (based the validation loss) will be stored in `~/.mltype/languages/cool_model`.

There are several important customizatons that one should be aware of.

Using MLflow

If one wants to get more training progress information theere is a flag `--use-mlflow` (requiring `mlflow` being installed). To launch the ui run the following commands

```
cd ~/.mltype/logs
mlflow ui
```

Multiple files

`mlt train` supports training from multiple files and folders. This is really useful if we want to recursively create a training set of all files in a given folder (e.g. github repository). Additionally, one can use the `--extensions` to control what files are considered when traversing a folder.

```
mlt train main.py folder_with_a_lot_of_files model --extensions ".py"
```

The above command will create a training set out of all files inside of the `folder_with_a_lot_of_files` folder having the ".py" suffix and also the *main.py*.

Excluding undesirable characters

If the input files contain some characters that we do not want the model to have in its vocabulary, we can simply use the `--illegal-chars` option. Internally, when an out of vocabulary character is encounter, there are two strategies to handle this (controled via `--fill-strategy`)

- **zeros** - vector of zeros is used
- **skip** - only consider samples that do not have out of vocabulary characters anywhere in their window

```
mlt train book.txt cool_model --illegal-chars "~{}`[]"
```

CHAPTER

THREE

EXAMPLES

3.1 Competing against yourself

CHAPTER

FOUR

MLTYPE PACKAGE

4.1 Submodules

4.2 mltype.base module

Building blocks.

class mltype.base.**Action**(*pressed_key*, *status*, *ts*)
Bases: object

Representation of one keypress.

Parameters

- **pressed_key** (*str*) – What key was pressed. We define a convention that pressing a backspace will be represented as *pressed_key=None*.
- **status** (*int*) – What was the status AFTER pushing the key. It should be one of the following integers:
  - STATUS_BACKSPACE
  - STATUS_CORRECT
  - STATUS_WRONG
- **ts** (*datetime*) – The timestamp corresponding to this action.

class mltype.base.**TypedText**(*text*)
Bases: object

Abstraction that represenets the text that needs to be typed.

Parameters **text** (*str*) – Text that needs to be typed.

actions
List of lists of Action instances of length equal to *len(text)*. It logs per character all actions that have been taken on it.

Type list

start_ts
Timestamp of when the first action was performed (not the time of initialization).

Type datetime or None

end_ts
Timestamp of when the last action was taken. Note that it is the action that lead to the text being correctly typed in it's entirity.

Type datetime or None

check_finished(*force_perfect=True*)
Determine whether the typing has been finished successfully.

Parameters **force_perfect** (*bool*) – If True, one can only finished if all the characters were typed correctly. Otherwise, all characters need to be either correct or wrong.

compute_accuracy()
Compute the accuracy of the typing.

compute_cpm()
Compute characters per minute.

compute_wpm(*word_size=5*)
Compute words per minute.

property elapsed_seconds
Get the number of seconds elapsed from the first action.

classmethod load(*path*)
Load a pickled file.

Parameters **path** (*pathlib.Path*) – Path to the pickle file.

Returns **typed_text** – Instance of the `TypedText`

Return type *TypedText*

property n_actions
Get the number of actions that have been taken.

property n_backspace_actions
Get the number of backspace actions.

property n_backspace_characters
Get the number of characters that have been backspaced.

property n_characters
Get the number of characters in the text.

property n_correct_characters
Get the number of characters that have been typed correctly.

property n_untouched_characters
Get the number of characters that have not been touched yet.

property n_wrong_characters
Get the number of characters that have been typed wrongly.

save(*path*)
Save internal state of this TypedText.

Can be loaded via the class method `load`.

Parameters **path** (*pathlib.Path*) – Where the .rlt file will be store.

type_character(*i*, *ch=None*)
Type one single character.

Parameters

- **i** (*int*) – Index of the character in the text.
- **ch** (*str or None*) – The character that was typed. Note that if None then we assume that the user used backspace.

unroll_actions()
Export actions in an order they appeared.

Returns **res** – List of tuples of *(ix_char, Action(..))*

Return type list

4.3 mltype.cli module

Command line interface.

4.4 mltype.data module

Data creating and managing.

mltype.data.**file2text**(*filepath*, *verbose=True*)
Read all lines of a file into a string.

Note that we destroy all the new line characters and all the whitespace charecters on both ends of the line. Note that this is very radical for source code of programming languages or similar.

Parameters

- **filepath** (*pathlib.Path*) – Path to the file
- **verbose** (*bool*) – If True, we print the name of the file.

Returns **text** – All the text found in the input file.

Return type str

mltype.data.**folder2text**(*folderpath*, *valid_extensions=None*)
Collect all files recursively and read into a list of strings.

4.5 mltype.interactive module

Module implementing interaction logic.

class mltype.interactive.**Cursor**(*stdscr*)
Bases: object

Utility class that can locate and modify the position of a cursor.

move_abs(*y*, *x*)
Move absolutely to cooordinates.

Note that if the column coordinate x is out of the screen then we automatically move to differnt row.

y, x [int] New coordinates where to move the cursor to.

class mltype.interactive.**Pen**(*font*, *background*, *i*)
Bases: object

Represents background and font color.

addch(*stdscr*, *y*, *x*, *text*)
Add a single character.

Parameters

- **stdscr** (*curses.Window*) – Window in which we add the character.
- **y** (*int*) – Position of the character.
- **x** (*int*) – Position of the character.
- **text** (*str*) – Single element string representing the character.

class mltype.interactive.**TypedTextWriter**(*tt*, *stdscr*, *y_start=0*, *x_start=0*, *replay_tt=None*, *target_wpm=None*)

Bases: object

Curses writer that uses the TypedText object.

We make an assumption that the x and y position of the starting character stay the same.

Parameters

- **tt** (TypedText) – Text that the user is going to type.
- **stdscr** (*curses.Window*) – Main curses window.
- **y_start** (*int*) – Coordinates of the first character.
- **x_start** (*int*) – Coordinates of the first character.
- **replay_tt** (TypedText *or None*) – If provided, it represents a previously typed text that we want to dynamically plot together with the current typing.

current_ix
Represents the index of the character of *self.tt.text* that we are about to type. Note this is exactly the character on which the cursor will be lying.

Type int

pens
The keys are integers representing different statuses. The values are *Pen* objects representing how to format a character with a given status. Note that if *replay_tt* is provided we add a new entry "replay" and it represents the style of replay character.

Type dict

replay_uactions
The unrolled actions of the replay.

Type list

replay_elapsed
The same length as *replay_uactions*. It stores the elapsed times (since the start) of all the actions. Note that it is going to be sorted in an ascending order and we can do binary search on it.

Type list

target_wpm
If specified, we display the uniform run that leads to that speed.

Type int or None

process_character()
Process an entered character.

render()
Render the entire screen.

property screen_status
Get screen information.

Returns

- **i_start** (*int*) – Integer representing the number of cells away from the start we are.
- **height, width** (*int*) – Height, width of the screen. Note that user my resize during a session.

mltype.interactive.**main_basic**(*text*, *force_perfect*, *output_file*, *instant_death*, *target_wpm*)
Run main curses loop with no previous replay.

Parameters

- **force_perfect** (*bool*) – If True, then one cannot finish typing before all characters are typed without any mistakes.
- **output_file** (*str or pathlib.Path or None*) – If `pathlib.Path` then we store the typed text in this file. If None, no saving is taking place.
- **instant_death** (*bool*) – If active, the first mistake will end the game.
- **target_wpm** (*int or None*) – The desired speed to be displayed as a guide.

mltype.interactive.**main_replay**(*replay_file*, *force_perfect*, *overwrite*, *instant_death*, *target_wpm*)
Run main curses loop with a replay.

Parameters **force_perfect** (*bool*) – If True, then one cannot finish typing before all characters are typed without any mistakes.

overwrite [bool] If True, the replay file will be overwritten in case we are faster than it.

replay_file [str or pathlib.Path] Typed text in this file from some previous game.

instant_death [bool] If active, the first mistake will end the game.

target_wpm [None or int] The desired speed to be shown as guide.

mltype.interactive.**run_loop**(*stdscr*, *text*, *force_perfect=True*, *replay_tt=None*, *instant_death=False*, *target_wpm=None*)
Run curses loop - actual implementation.

4.6 mltype.ml module

Machine learning utilities.

class mltype.ml.**LanguageDataset**(*X*, *y*, *vocabulary*, *transform=None*)
Bases: `torch.utils.data.dataset.Dataset`

Language dataset.

All the inputs of this class should be generated via *create_data_language*.

Parameters

- **X** (*np.ndarray*) – Array of shape (n_samples, window_size) of dtype *np.int8*. It represents the features.
- **y** (*np.ndarray*) – Array of shape (n_samples,) of dtype *np.int8*. It represents the targets
- **vocabulary** (*list*) – List of characters in the vocabulary.

- **transform** (*callable or None*) – Some callable that inputs *X* and *y* and returns some modified instances of them.

ohv_matrix
Matrix of shape *(vocab_size + 1, vocab_size)*. The submatrix *ohv_matrix[:vocab_size, :]* is an identity matrix and is used for fast creation of one hot vectors. The last row of *ohv_matrix* is a zero vector and is used for encoding out-of-vocabulary characters.

Type np.ndarray

class mltype.ml.**SingleCharacterLSTM**(*vocab_size*, *hidden_size=16*, *n_layers=1*, *dense_size=128*)
Bases: pytorch_lightning.core.lightning.LightningModule

Single character recurrent neural network.

Given some string of characters, we generate the probability distribution of the next character.

Architecture starts with an LSTM (*hidden_size*, *n_layers*, *vocab_size*) network and then we feed the last hidden state to a fully connected network with one hidden layer (*dense_size*).

Parameters

- **vocab_size** (*int*) – Size of the vocabulary. Necessary since we are encoding each character as a one hot vector.
- **hidden_size** (*int*) – Hidden size of the recurrent cell.
- **n_layers** (*int*) – Number of layers in the recurrent network.
- **dense_size** (*int*) – Size of the single layer of the feed forward network.

rnn_layer
The recurrent network layer.

Type torch.nn.Module

linear_layer1
Linear layer connecting the last hidden state and the single layer of the feedforward network.

Type torch.nn.Module

linear_layer2
Linear layer connecting the single layer of the feedforward network with the output (of size *vocabulary_size*).

Type torch.nn.Module

activation_layer
Softmax layer making sure we get a probability distribution.

Type torch.nn.Module

configure_optimizers()
Configure optimizers.

Necessary for pytorch-lightning.

Returns **optimizer** – The chosen optimizer.

Return type Optimizer

forward(*x*, *h=None*, *c=None*)
Perform forward pass.

Parameters

- **x** (*torch.Tensor*) – Input features of shape *(batch_size, window_size, vocab_size)*. Note that the provided *vocab_size* needs to be equal to the one provided in the constructor. The remaining dimensions (*batch_size* and *window_size*) can be any positive integers.
- **h** (*torch.Tensor*) – Hidden states of shape *(n_layers, batch_size, hidden_size)*. Note that if provided we enter a continuation mode. In this case to generate the prediction we just use the last character and the hidden state for the prediction. Note that in this case we enforce that *x.shape=(batch_size, 1, vocab_size)*.
- **c** (*torch.Tensor*) – Hidden states of shape *(n_layers, batch_size, hidden_size)*. Note that if provided we enter a continuation mode. In this case to generate the prediction we just use the last character and the hidden state for the prediction. Note that in this case we enforce that *x.shape=(batch_size, 1, vocab_size)*.

Returns

- **probs** (*torch.Tensor*) – Tensor of shape *(batch_size, vocab_size)*. For each sample it represents the probability distribution over all characters in the vocabulary.
- **h_n, c_n** (*torch.Tensor*) – New Hidden states of shape *(n_layers, batch_size, hidden_size)*.

training: bool

training_step(*batch*, *batch_idx*)
Run training step.

Necessary for pytorch-lightning.

Parameters

- **batch** (*tuple*) – Batch of training samples. The exact definition depends on the dataloader.
- **batch_idx** (*idx*) – Index of the batch.

Returns **loss** – Tensor scalar representing the mean binary cross entropy over the batch.

Return type torch.Tensor

validation_epoch_end(*outputs*)
Run epoch end validation logic.

We sample 5 times 100 characters from the current network. We then print to the standard output.

Parameters **outputs** (*list*) – List of batches that were collected over the validation set with *validation_step*.

validation_step(*batch*, *batch_idx*)
Run validation step.

Optional for pytorch-lightning.

Parameters **batch** (*tuple*) – Batch of validation samples. The exact definition depends on the dataloader.

batch_idx [idx] Index of the batch.

Returns **vocabulary** – Vocabulary in order to have access in *validation_epoch_end*.

Return type list

mltype.ml.**create_data_language**(*text*, *vocabulary*, *window_size=2*, *fill_strategy='zeros'*, *verbose=False*)
Create a supervised dataset for the characte/-lever language model.

Parameters

- **text** (*str*) – Some text.
- **vocabulary** (*list*) – Unique list of supported characters. Their corresponding indices are going to be used for the one hot encoding.
- **window_size** (*int*) – The number of previous characters to condition on.
- **fill_strategy** (*str, {"skip", "zeros"}*) – Strategy for handling initial characters and unknown characters.
- **verbose** (*bool*) – If True, progress bar is showed.

Returns

- **X** (*np.ndarray*) – Features array of shape *(len(text), window_size)* if *fill_strategy=zeros*, otherwise it might be shorter. The dtype is *np.int8*. If applicable, the integer *(len(vocabulary))* represnts a zero vector (out of vocabulary token).
- **y** (*np.ndarray*) – Targets array of shape *(len(text),)* if *fill_strategy=zeros*, otherwise it might be shorter. The dtype is *np.int8*.
- **indices** (*np.ndarray*) – For each sample an index of the character we are trying to predict. Note that for *fill_strategy="zeros"* it is going to be *np.arange(len(text))*. However, for different strategies might have gaps. It helps us to keep track of the sample - character correspondence.

mltype.ml.**load_model**(*path*)

Load serialized model and vocabulary.

Parameters **path** (*pathlib.Path*) – Path to where the file lies. This file was created by *save_model* method.

Returns

- **model_inst** (*SingleCharacterLSTM*) – Instance of the model. Note that all of its parameters will be lying on a CPU.
- **vocabulary** (*list*) – Corresponding vocabulary.

mltype.ml.**run_train**(*texts*, *name*, *max_epochs=10*, *window_size=50*, *batch_size=32*, *vocab_size=None*, *fill_strategy='skip'*, *illegal_chars=''*, *train_test_split=0.5*, *hidden_size=32*, *dense_size=32*, *n_layers=1*, *checkpoint_path=None*, *output_path=None*, *use_mlflow=True*, *early_stopping=True*, *gpus=None*)

Run the training loop.

Note that the parameters are also explained in the cli of *mlt train*.

Parameters

- **texts** (*list*) – List of str representing all texts we would like to train on.
- **name** (*str*) – Name of the model. This name is only used when we save the model - it is not hardcoded anywhere in the serialization.
- **max_epochs** (*int*) – Maximum number of epochs. Note that the number of actual epochs can be lower if we activate the *early_stopping* flag.
- **window_size** (*int*) – Number of previous characters to consider when predicting the next character. The higher the number the longer the memory we are enforcing. Howerever, at the same time, the training becomes slower.
- **batch_size** (*int*) – Number of samples in one batch.

- **vocab_size** (*int*) – Maximum number of characters to be put in the vocabulary. Note that one can explicityly exclude characters via *illegal_chars*. The higher this number the bigger the feature vectors are and the slower the training.
- **fill_strategy** (*str, {"zeros", "skip"}*) – Determines how to deal with out of vocabulary characters. When "zeros" then we simply encode them as zero vectors. If "skip", we skip a given sample if any of the characters in the window or the predicted character are not in the vocabulary.
- **illegal_chars** (*str or None*) – If specified, then each character of the str represents a forbidden character that we do not put in the vocabulary.
- **train_test_split** (*float*) – Float in the range (0, 1) representing the percentage of the training set with respect to the entire dataset.
- **hidden_size** (*int*) – Hidden size of LSTM cells (equal in all layers).
- **dense_size** (*int*) – Size of the dense layer that is bridging the hidden state outputted by the LSTM and the final output probabilities over the vocabulary.
- **n_layers** (*int*) – Number of layers inside of the LSTM.
- **checkpoint_path** (*None or pathlib.Path or str*) – If specified, it is pointing to a checkpoint file (generated by Pytorch-lightning). This file does not contain the vocabulary. It can be used to continue the training.
- **output_path** (*None or pathlib.Path or str*) – If specified, it is an alternative output folder when the trained models and logging information will be stored. If not specified the output folder is by default set to *~/.mltype*.
- **use_mlflow** (*bool*) – If active, than we use mlflow for logging of training and validation loss. Additionally, at the end of each epoch we generate a few sample texts to demonstrate how good/bad the current network is.
- **early_stopping** (*bool*) – If True, then we monitor the validation loss and if it does not improve for a certain number of epochs then we stop the traning.
- **gpus** (*int or None*) – If None or 0, no GPUs are used (only CPUs). Otherwise, it represents the number of GPUs to be used (using the data parallelization strategy).

mltype.ml.**sample_char**(*network*, *vocabulary*, *h=None*, *c=None*, *previous_chars=None*, *random_state=None*, *top_k=None*, *device=None*)

Sample a character given network probability prediciton (with a state).

Parameters

- **network** (*torch.nn.Module*) – Trained neural network that outputs a probability distribution over *vocabulary*.
- **vocabulary** (*list*) – List of unique characters.
- **h** (*torch.Tensor*) – Hidden states with shape *(n_layers, batch_size=1, hidden_size)*. Note that if both of them are None we are at the very first character.
- **c** (*torch.Tensor*) – Hidden states with shape *(n_layers, batch_size=1, hidden_size)*. Note that if both of them are None we are at the very first character.
- **previous_chars** (*None or str*) – Previous charaters. None or and empty string if we are at the very first character.
- **random_state** (*None or int*) – Guarantees reproducibility.
- **top_k** (*None or int*) – If specified, we only sample from the top k most probably characters. Otherwise all of them.

- **device** (*None or torch.device*) – By default *torch.device("cpu")*.

Returns **ch** – A character from the vocabulary.

Return type str

mltype.ml.**sample_text**(*n_chars*, *network*, *vocabulary*, *initial_text=None*, *random_state=None*, *top_k=None*, *verbose=False*, *device=None*)

Sample text by unrolling character by character predictions.

Note that keep the pass hidden states with each character prediciton and there is not need to specify a window.

Parameters

- **n_chars** (*int*) – Number of characters to sample.
- **network** (*torch.nn.Module*) – Pretrained character level network.
- **vocabulary** (*list*) – List of unique characters.
- **initial_text** (*None or str*) – If specified, initial text to condition based on.
- **random_state** (*None or int*) – Allows reproducibility.
- **top_k** (*None or int*) – If specified, we only sample from the top k most probable characters. Otherwise all of them.
- **verbose** (*bool*) – Controls verbosity.
- **device** (*None or torch.device*) – By default *torch.device("cpu")*.

Returns **text** – Generated text of length *n_chars* + *len(initial_text)*.

Return type str

mltype.ml.**save_model**(*model*, *vocabulary*, *path*)

Serialize a model.

Note that we require that the model has a property *hparams* that we can unpack into the constructor of the class and get the same network architecture. This is automatically the case if we subclass from *pl.LightningModule*.

Parameters

- **model** (SingleCharacterLSTM) – Torch model to be saved. Additionally, we require that it has the *hparams* property that contains all necessary hyperparameters to instantiate the model.
- **vocabulary** (*list*) – The corresponding vocabulary.
- **path** (*pathlib.Path*) – Path to the file that will whole the serialized object.

mltype.ml.**text2features**(*text*, *vocabulary*)

Create per character one hot encoding.

Note that we employ the zeros strategy out of vocabulary characters.

Parameters

- **text** (*str*) – Text.
- **vocabulary** (*list*) – Vocabulary to be used for the endcoding.

Returns **res** – Array of shape *(len(text), len(vocabulary)* of boolean dtype. Each row represents the one hot encoding of the respective character. Note that out of vocabulary characters are encoding with a zero vector.

Return type np.ndarray

4.7 mltype.stats module

Computation of various statistics.

mltype.stats.**times_per_character**(*tt*)
Compute per caracter analysis.

Parameters **tt** (TypedText) – Instance of the TypedText.

Returns **stats** – Keys are characters and values are list of time intervals it took to write the last correct instance.

Return type dict

4.8 mltype.utils module

Collection of utility functions.

mltype.utils.**get_cache_dir**(*predefined_path=None*)
Get the cache directory path and potentially create it.

If no predefined path provided, we simply take *~/.mltype*. Note that if one changes the *os.environ["home"]* dynamically it will influence the output of this function. this is done on purpose to simplify testing.

Parameters **predefined_path** (*None or pathlib.Path or str*) – If provided, we just return the same path. We potentially create the directory if it does not exist. If it is not provided we use *$HOME/.mltype*.

Returns **path** – Path to where the caching directory is located.

Return type pathlib.Path

mltype.utils.**get_mlflow_artifacts_path**(*client*, *run_id*)
Get path to where the artifacts are located.

The benefit is that we can log any file into it and even create a custom folder hierarachy.

Parameters

- **client** (*mlflow.tracking.MlflowClient*) – Client.
- **run_id** (*str*) – Unique identifier of a run.

Returns **path** – Path to the root folder of artifacts.

Return type pathlib.Path

mltype.utils.**print_section**(*name*, *fill_char='='*, *drop_end=False*, *add_ts=True*)
Print nice section blocks.

Parameters

- **name** (*str*) – Name of the section.
- **fill_char** (*str*) – Character to be used for filling the line.
- **drop_end** (*bool*) – If True, the ending line is not printed.
- **add_ts** (*bool*) – We add a time step to the heading.

4.9 Module contents

Python package.

PYTHON MODULE INDEX

m

INDEX

O

P

R

S

T

U

V