CAS Advanced Machine Learning 2022

University of Bern

American Sign Language Recognition with DeepLearning

Project documentation presented by

Felix Schlatter
felix.schlatter@bluemail.ch

Asad Bin Imtiaz
asad.imtiaz@students.unibe.ch

Table of contents

Chapter 1

Data Augmentations

This module contains functions for data augmentation and normalization of hand landmarks extracted from sign language videos.

Table 1: Summary of Augmentations

Functions	Description
shift_landmarks(frames, max_shift=0.01)	Shifts the landmark coordinates randomly by a small amount.
mirror_landmarks(frames)	Inverts (mirrors) the landmark coordinates along the x-axis.
frame_dropout(frames, dropout_rate=0.05)	Randomly drops frames from the input landmark data based on a specified dropout rate.
random_scaling(frames, scale_range=(0.9, 1.1))	Applies random scaling to the landmark coordinates within a specified scale range.
random_rotation(frames, max_angle=10)	Applies a random rotation to the landmark coordinates, with the rotation angle within a specified range.
normalize(frames, mn, std)	Normalizes the frames using a given mean and standard deviation.
standardize(frames)	Standardizes the frames so that they have a mean of 0 and a standard deviation of 1.

`augmentations.frame_dropout(`*frames*, *dropout_rate=0.05*`)`

Randomly drop frames from the input landmark data.

Args:
frames (numpy.ndarray): An array of landmarks data. dropout_rate (float): The proportion of frames to drop (default: 0.05).

Returns:
numpy.ndarray: An array of landmarks with dropped frames.

`augmentations.mirror_landmarks(`*frames*`)`

Invert/mirror landmark coordinates along the x-axis.

Args:
frames (numpy.ndarray): An array of landmarks data.

Returns:
numpy.ndarray: An array of inverted landmarks.

`augmentations.normalize(`*frames*, *mn*, *std*`)`

Normalize the frames with a given mean and standard deviation.

Args:
frames (numpy.ndarray): An array of landmarks data. mn (float): The mean value for normalization. std (float): The standard deviation for normalization.

Returns:
numpy.ndarray: An array of normalized landmarks.

`augmentations.random_rotation`(*frames*, *max_angle=10*)

Apply random rotation to landmark coordinates. (on X and Y only)

Args:
frames (numpy.ndarray): An array of landmarks data. max_angle (int): The maximum rotation angle in degrees (default: 10).

Returns:
numpy.ndarray: An array of landmarks with randomly rotated coordinates.

`augmentations.random_scaling`(*frames*, *scale_range=(0.9, 1.1)*)

Apply random scaling to landmark coordinates.

Args:
frames (numpy.ndarray): An array of landmarks data. scale_range (tuple): A tuple containing the minimum and maximum scaling factors (default: (0.9, 1.1)).

Returns:
numpy.ndarray: An array of landmarks with randomly scaled coordinates.

`augmentations.shift_landmarks`(*frames*, *max_shift=0.01*)

Shift landmark coordinates randomly by a small amount.

Args:
frames (numpy.ndarray): An array of landmarks data. max_shift (float): Maximum shift for the random shift (default: 0.01).

Returns:
numpy.ndarray: An array of augmented landmarks.

`augmentations.standardize`(*frames*)

Standardize the frames so that they have mean 0 and standard deviation 1.

Args:
frames (numpy.ndarray): An array of landmarks data.

Returns:
numpy.ndarray: An array of standardized landmarks.

Chapter 2

Training Callbacks

1 Callbacks description

This module contains callback codes which may be executed during training. These callbacks are used to dynamically adjust the dropout rate and data augmentation probability during the training process, which can be useful techniques to prevent overfitting and increase the diversity of the training data, potentially improving the model's performance.

The dropout_callback function is designed to increase the dropout rate of the model during the training process after a certain number of epochs. The dropout rate is a regularization technique used to prevent overfitting during the training process. The rate of dropout is increased every few epochs based on a specified rate until it reaches a specified maximum limit.

The augmentation_increase_callback: function is designed to increase the probability of data augmentation applied to the dataset during the training process after a certain number of epochs. Data augmentation is a technique that can generate new training samples by applying transformations to the existing data. The probability of data augmentation is increased every few epochs based on a specified rate until it reaches a specified maximum limit.

`callbacks.augmentation_increase_callback(`*trainer*, *aug_increase_rate=1.5*, *max_limit=0.6*`)`

A callback function designed to increase the probability of data augmentation applied on the dataset during the training process. Data augmentation is a technique that can generate new training samples by applying transformations to the existing data.

The increase in data augmentation is performed every few epochs based on 'DYNAMIC_AUG_INC_INTERVAL' until it reaches a specified maximum limit.

Args:

trainer: The object that contains the model and handles the training process. aug_increase_rate: The rate at which data augmentation probability is increased. Default is value of 'DYNAMIC_AUG_INC_RATE' from config. max_limit: The maximum limit to which data augmentation probability can be increased. Default is value of 'DYNAMIC_AUG_MAX_THRESHOLD' from config.

Returns:

None

Functionality:

Increases the probability of data augmentation applied on the dataset after certain number of epochs defined by 'DYNAMIC_AUG_INC_INTERVAL'.

Parameters

- `trainer` – Trainer object handling the training process
- `aug_increase_rate` – Rate at which to increase the data augmentation probability
- `max_limit` – Maximum allowable data augmentation probability

Returns

None

Return type
None

`callbacks.dropout_callback(`*trainer*, *dropout_ rate=1.1*, *max_ dropout=0.35*`)`

A callback function designed to increase the dropout rate of the model in training after a certain number of epochs. The dropout rate is a regularization technique which helps in preventing overfitting during the training process.

The rate of dropout is increased every few epochs based on the config parameter (in config.py) 'DYNAMIC_DROP_OUT_REDUCTION_INTERVAL' until a maximum threshold defined by 'max_dropout'. This function is usually called after each epoch in the training process.

Args:
trainer: The object that contains the model and handles the training process. dropout_rate: The rate at which the dropout rate is increased. Default is value of 'DYNAMIC_DROP_OUT_REDUCTION_RATE' from config. max_dropout: The maximum limit to which dropout can be increased. Default is value of 'DYNAMIC_DROP_OUT_MAX_THRESHOLD' from config.

Returns:
None

Functionality:
Increases the dropout rate of all nn.Dropout modules in the model after certain number of epochs defined by 'DYNAMIC_DROP_OUT_REDUCTION_INTERVAL'.

Parameters
- `trainer` – Trainer object handling the training process
- `dropout_rate` – Rate at which to increase the dropout rate
- `max_dropout` – Maximum allowable dropout rate

Returns
None

Return type
None

Chapter 3

Project Configuration

1 Project configuration description

This configuration is created allows for easy tuning of your machine learning model's parameters and setup. The device on which the model runs, the paths for various resources, the seed for random number generation, hyperparameters for model training, and much more and quickly be change and configured. This makes your setup flexible and easy to adapt for various experiments and environments

`config.BATCH_SIZE = 128`

Training Batch Size

`config.CHECKPOINT_DIR = 'checkpoints/'`

Checkpoint files Directory path

`config.CLEANED_FILE = 'cleansed_data.marker'`

File that marks the data cleaning stage.

`config.COLUMNS_TO_USE = ['x', 'y']`

Coordinate columns from the data to use for training.

`config.DATA_DIR = 'data/'`

Data files Directory path

`config.DEVICE = 'cpu'`

Setting the device for training, 'cuda' if a CUDA-compatible GPU is available, 'mps' if multiple processors are available, 'cpu' if none of the above.

`config.DL_FRAMEWORK = 'pytorch'`

Deep learning framework to use for training and inference. Can be either 'pytorch' or 'tensorflow'.

`config.DYNAMIC_AUG_INC_INTERVAL = 5`

The number of epochs to wait before increasing the probability of data augmentation.

`config.DYNAMIC_AUG_INC_RATE = 1.5`

The rate at which the probability of data augmentation is increased.

`config.DYNAMIC_AUG_MAX_THRESHOLD = 0.6`

The maximum limit to which the probability of data augmentation can be increased.

`config.DYNAMIC_DROP_OUT_INIT_RATE = 0.01`

The value of initial low dropouts rate

`config.DYNAMIC_DROP_OUT_MAX_THRESHOLD = 0.35`

The max value of dynamic dropouts

`config.DYNAMIC_DROP_OUT_REDUCTION_INTERVAL = 2`

The epoch interval value to gradually change dropout rate

`config.DYNAMIC_DROP_OUT_REDUCTION_RATE = 1.1`

The value to increase dropouts by

`config.EARLY_STOP_METRIC = 'accuracy'`

Which metric should be used for early stopping loss/accuracy

`config.EARLY_STOP_MODE = 'max'`

What is the mode? min/max

`config.EARLY_STOP_PATIENCE = 5`

The number of epochs to wait for improvement in the validation loss before stopping training

`config.EARLY_STOP_TOLERENCE = 0.001`

The value of loss as margin to tolerate

`config.EPOCHS = 60`

Training Number of epochs

`config.FACE_FEATURES = 468`

Number of features related to the face in the data.

`config.FACE_FEATURE_START = 0`

Start index for face feature in the data.

`config.FACE_INDICES = array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39], dtype=int64)`

Indices of face landmarks that are used from the data.

`config.FACE_LANDMARKS = array([ 61, 185, 40, 39, 37, 0, 267, 269, 270, 409, 291, 146, 91, 181, 84, 17, 314, 405, 321, 375, 78, 191, 80, 81, 82, 13, 312, 311, 310, 415, 95, 88, 178, 87, 14, 317, 402, 318, 324, 308])`

Landmarks for Lips

`config.HAND_FEATURES = 21`

Number of features related to the hand in the data.

`config.HAND_INDICES = array([40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81], dtype=int64)`

Indices of hand landmarks that are used from the data.

`config.INPUT_SIZE = 32`

Size of the input data for the model.

`config.INTEREMOLATE_MISSING = 3`

Number of missing values to interpolate in the data.

`config.LANDMARK_FILES = 'train_landmark_files'`

Directory where training landmark files are stored.

`config.LEARNING_RATE = 0.001`

Training Learning rate

`config.LEFT_HAND_FEATURE_START = 468`

Start index for left hand feature in the data.

`config.LEFT_HAND_INDICES = array([40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60], dtype=int64)`

Indices of left hand landmarks that are used from the data.

`config.LIMIT_BATCHES = 3`

Number of batches to run (Only active if FAST_DEV_RUN is set to True)

`config.LIMIT_EPOCHS = 1`

Number of Epochs to run (Only active if FAST_DEV_RUN is set to True)

`config.LOG_METRICS = ['Accuracy', 'Loss', 'F1Score', 'Precision', 'Recall']`

> **Warning:** Training/Validation/Testing will only be done on LIMIT_BATCHES and LIMIT_EPOCHS, if FAST_DEV_RUN is set to True

`config.MAP_JSON_FILE = 'sign_to_prediction_index_map.json'`

JSON file that maps sign to prediction index.

`config.MARKER_FILE = 'preprocessed_data.marker'`

File that marks the preprocessing stage.

`config.MAX_SEQUENCES = 32`

Maximum number of sequences in the input data.

`config.MIN_SEQUEENCES = 8.0`

Minimum number of sequences in the input data.

`config.MODELNAME = 'LSTMPredictor'`

Name of the model to be used for training.

`config.MODEL_DIR = 'models/'`

Model files Directory path

`config.N_CLASSES = 250`

Number of classes

`config.N_DIMS = 2`

Number of dimensions used in training

`config.N_LANDMARKS = 96`

Total number of used landmarks

`config.OUT_DIR = 'out/'`

Output files Directory path

`config.POSE_FEATURES = 33`

Number of features related to the pose in the data.

`config.POSE_FEATURE_START = 489`

Start index for pose feature in the data.

`config.POSE_INDICES = array([82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95], dtype=int64)`

Indices of pose landmarks that are used from the data.

`config.PROCESSED_DATA_DIR = 'data/processed/'`

Processed Data files Directory path

`config.RAW_DATA_DIR = 'data/raw/'`

Raw Data files Directory path

`config.RIGHT_HAND_FEATURE_START = 522`

Start index for right hand feature in the data.

`config.RIGHT_HAND_INDICES = array([61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81], dtype=int64)`

Indices of right hand landmarks that are used from the data.

`config.ROOT_PATH = 'C:\\Users\\fs.GUNDP\\Python\\CAS-AML-FINAL-PROJECT\\docs\\../src\\../'`

Root directory

`config.ROWS_PER_FRAME = 543`

Number of rows per frame in the data.

`config.RUNS_DIR = 'runs/'`

Run files Directory path

`config.SEED = 0`

Set Random Seed

`config.SKIP_CONSECUTIVE_ZEROS = 4`

Skip data if there are this many consecutive zeros.

`config.SRC_DIR = 'src/'`

Source files Directory path

`config.TEST_SIZE = 0.05`

Testing Test set size

`config.TRAIN_CSV_ADDON_FILE = 'train_add.csv'`

CSV file name that contains the additional training dataset from videos.

`config.TRAIN_CSV_FILE = 'train.csv'`

CSV file name that contains the training dataset.

`config.TRAIN_SIZE = 0.9`

Training Train set split size

`config.TUNE_HP = True`

Tune hyperparameters

`config.USED_FACE_FEATURES = 40`

Count of facial features used

`config.USED_HAND_FEATURES = 21`

Count of hands features used (single hand only)

`config.USED_POSE_FEATURES = 14`

Count of body/pose features used

`config.USEFUL_ALL_LANDMARKS = array([61, 185, 40, 39, 37, 0, 267, 269, 270, 409, 291, 146, 91, 181, 84, 17, 314, 405, 321, 375, 78, 191, 80, 81, 82, 13, 312, 311, 310, 415, 95, 88, 178, 87, 14, 317, 402, 318, 324, 308, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513])`

All Landmarks

`config.USEFUL_FACE_LANDMARKS = array([ 61, 185, 40, 39, 37, 0, 267, 269, 270, 409, 291, 146, 91, 181, 84, 17, 314, 405, 321, 375, 78, 191, 80, 81, 82, 13, 312, 311, 310, 415, 95, 88, 178, 87, 14, 317, 402, 318, 324, 308])`

Landmarks for face

```
config.USEFUL_HAND_LANDMARKS = array([468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478,
479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 522, 523, 524, 525, 526, 527, 528, 529, 530,
531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542])
```

Landmarks for both hands

```
config.USEFUL_LEFT_HAND_LANDMARKS = array([468, 469, 470, 471, 472, 473, 474, 475, 476, 477,
478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488])
```

Landmarks for left hand

```
config.USEFUL_POSE_LANDMARKS = array([500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510,
511, 512, 513])
```

Landmarks for pose

```
config.USEFUL_RIGHT_HAND_LANDMARKS = array([522, 523, 524, 525, 526, 527, 528, 529, 530, 531,
532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542])
```

Landmarks for right hand

```
config.VALID_SIZE = 0.05
```

Training Validation set size

Chapter 4

Data Utilities

1 Data processing Utils description

This module handles the loading and preprocessing of data. It is specifically tailored for loading ASL sign language dataset where the raw data includes information about the position of hands, face, and body over time.

ASL stands for American Sign Language, which is a natural language used by individuals who are deaf or hard of hearing to communicate through hand gestures and facial expressions.

The dataset consists of sequences of frames, where each frame contains multiple "landmarks". Each of these landmarks has multiple features, such as coordinates. The landmarks may represent various aspects of human body, such as facial features, hand positions, and body pose.

This module is used to process the raw data, to create a uniform dataset where all sequences are of the same length and all missing values have been handled in a way that maintains the integrity of the data. This involves steps like detecting and removing empty frames, selecting specific landmarks, resizing sequences and handling NaN values.

`data.data_utils.calculate_avg_landmark_positions(`*dataset*`)`

Calculate the average landmark positions for left-hand, right-hand, and face landmarks for each sign in the dataset. The purpose of this function is to compute the average positions of landmarks for left-hand, right-hand, and face for each sign in the training dataset.

Returns: List : Containing a dictionary with average x/y positions with keys - 'left_hand' - 'right_hand' - 'face'

Functionality: - The function takes an ASLDataset object as an input, which contains the training data. - It calculates the average landmark positions for left-hand, right-hand, and face landmarks for each sign in the dataset. - The function returns a list containing a dictionary with average x/y positions with keys 'left_hand', 'right_hand', and 'face' for each sign.

Parameters

`dataset` (`ASL_DATASET`) – The ASL dataset object containing the training data.

Returns

A list containing a dictionary with average x/y positions with keys 'left_hand', 'right_hand', and 'face'

for each sign. :rtype: List[Dict[str, np.ndarray]]

`data.data_utils.calculate_landmark_length_stats()`

Calculate statistics of landmark lengths for each sign type.

Returns: dict: A dictionary of landmark lengths for each sign type containing: - minimum - maximum - mean - median - standard deviation

Functionality: - The function reads the CSV file. - It groups the DataFrame by sign. - An empty dictionary is created to store average landmarks for each sign type. - The function loops through each unique sign and its corresponding rows in the grouped DataFrame. - For each sign, it initializes a list to store the length of landmarks for each example of the current sign. - It loops through each row of the current sign type, loads the data, and adds the length of landmarks of the current example to the list of current sign data. - The function calculates the minimum, maximum, mean, standard deviation, and median of the landmarks for the current sign and updates the dictionary. - The resulting dictionary containing average landmarks for each sign type is returned.

Returns
A dictionary of landmark lengths for each sign type containing minimum, maximum, mean, median & standard

deviation :rtype: dict

`data.data_utils.create_data_loaders(`*asl_dataset*, *train_size=0.9*, *valid_size=0.05*, *test_size=0.05*, *batch_size=128*, *random_state=0*, *dl_framework='pytorch'*, *num_workers=16*`)`

Split the ASL dataset into training, validation, and testing sets and create data loaders for each set.

Args: asl_dataset (ASLDataset): The ASL dataset to load data from. train_size (float, optional): The proportion of the dataset to include in the training set. Defaults to 0.8. valid_size (float, optional): The proportion of the dataset to include in the validation set. Defaults to 0.1. test_size (float, optional): The proportion of the dataset to include in the testing set. Defaults to 0.1. batch_size (int, optional): The number of samples per batch to load. Defaults to BATCH_SIZE. random_state (int, optional): The seed used by the random number generator for shuffling the data. Defaults to SEED.

Returns: tuple of DataLoader: A tuple containing the data loaders for training, validation, and testing sets.

Parameters

- `asl_dataset` (*ASLDataset*) – The ASL dataset to load data from.
- `train_size` (*float*) – The proportion of the dataset to include in the training set.
- `valid_size` (*float*) – The proportion of the dataset to include in the validation set.
- `test_size` (*float*) – The proportion of the dataset to include in the testing set.
- `batch_size` (*int*) – The number of samples per batch to load.
- `random_state` (*int*) – The seed used by the random number generator for shuffling the data.

Returns
A tuple containing the data loaders for training, validation, and testing sets.

Return type
tuple of DataLoader

`data.data_utils.interpolate_missing_values(`*arr*, *max_gap=3*`)`

This function provides a solution for handling missing values in the data array. It interpolates these missing values, filling them with plausible values that maintain the overall data integrity. The function uses a linear interpolation method that assumes a straight line between the two points on either side of the gap. The maximum gap size for which interpolation should be performed is also configurable.

Arguments: The function takes two arguments - an array with missing values, and a maximum gap size for interpolation. If the size of the gap (i.e., number of consecutive missing values) is less than or equal to this specified maximum gap size, the function will fill it with interpolated values. This ensures that the data maintains its continuity without making too far-fetched estimations for larger gaps.

Args:
arr (np.ndarray): Input array with missing values. max_gap (int, optional): Maximum gap to fill. Defaults to INTEREMOLATE_MISSING.

Returns:
np.ndarray: Array with missing values interpolated.

Functionality:
Interpolates missing values in the array. The function fills gaps of up to a maximum size with interpolated values, maintaining data integrity and continuity.

Returns
Array with missing values interpolated.

Return type
: np.ndarray

Parameters
- `arr` (*np.ndarray*) – Input array with missing values.
- `max_gap` (*int*) – Maximum gap to fill.

This function uses linear interpolation to fill the missing values. Other forms of interpolation such as polynomial or spline may provide better results for specific types of data. It is also worth noting that no imputation method can fully recover original data, and as such, results should be interpreted with caution when working with imputed data.

`data.data_utils.load_relevant_data_subset(`*pq_path*`)`

This function serves a key role in handling data in our pipeline by loading only a subset of the relevant data from a given path. The primary purpose of this is to reduce memory overhead when working with large datasets. The implementation relies on efficient data loading strategies, leveraging the speed of Parquet file format and the ability to read in only necessary chunks of data instead of the whole dataset.

The function takes as input a string which represents the path to the data file. It makes use of pandas' parquet read function to read the data file. This function is particularly suited for reading large datasets as it allows for efficient on-disk storage and fast query capabilities. The function uses PyArrow library as the engine for reading the parquet files which ensures efficient and fast reading of data. After reading the data, the function selects the relevant subset based on certain criteria, which is task specific.

Args:
: pq_path (str): Path to the data file.

Returns:
: np.ndarray: Subset of the relevant data as a NumPy array.

Functionality:
: Loads a subset of the relevant data from a given path.

Returns
: Subset of the relevant data.

Return type
: np.ndarray

Parameters
: `pq_path` (*str*) – Path to the data file.

The function assumes that the data file is in parquet format and the necessary libraries for reading parquet files are installed. It also assumes that the path provided is a valid path to the data file.

`data.data_utils.preprocess_data(`*landmarks*`)`

This function preprocesses the input data by applying similar steps as the preprocess_data_to_same_size function, but with the difference that it does not interpolate missing values. The function again targets to adjust the size of the input data to align with the INPUT_SIZE. It selects only non-empty frames and follows similar strategies of padding, repeating, and pooling the data for size alignment.

Args:
: landmarks (np.ndarray): The input array with landmarks data.

Returns:
: Tuple[np.ndarray, int]: A tuple containing processed landmark data and the final size of the data.

Parameters
: `landmarks` (*np.ndarray*) – The input array with landmarks data.

Returns
: A tuple containing processed landmark data and the final size of the data.

Return type
: Tuple[np.ndarray, int]

`data.data_utils.preprocess_data_item(`*raw_ landmark_path, targets_ sign*`)`

The function preprocesses landmark data for a single file. The process involves applying transformations to raw landmark data to convert it into a form more suitable for machine learning models. The transformations may include normalization, scaling, etc. The target sign associated with the landmark data is also taken as input.

This function is a handy function to process all landmark aequences on a particular location. This will come in handy while testing where individual sequences may be provided

Args:

raw_landmark_path: Path to the raw landmark file targets_sign: The target sign for the given landmark data

Returns: dict: A dictionary containing the preprocessed landmarks, target, and size.

Functionality: - The function reads the parquet file and processes the data. - It filters columns to include only frame, type, landmark_index, x, and y. - The function then filters face mesh landmarks and pose landmarks based on the predefined useful landmarks. - Landmarks data is pivoted to have a multi-level column structure on landmark type and frame sequence ids. - Missing values are interpolated using linear interpolation, and any remaining missing values are filled with 0. - The function rearranges columns and calculates the number of frames in the data. - X and Y coordinates are brought together, and a dictionary with the processed data is created and returned.

Parameters

- `raw_landmark_path` (*str*) – Path to the raw landmark file.
- `targets_sign` (*int*) – The target sign for the given landmark data.

Returns

A dictionary containing the preprocessed landmarks, target, and size.

Return type

dict

`data.data_utils.preprocess_data_to_same_size(`*landmarks*`)`

This function preprocesses the input data to ensure all data arrays have the same size, specified by the global INPUT_SIZE variable. This uniform size is necessary for subsequent processing and analysis stages, particularly those involving machine learning models which often require consistent input sizes. The preprocessing involves several steps, including handling missing values, upsampling, and reshaping arrays. It begins by interpolating any missing values, and then it subsets the data by selecting only non-empty frames. Various strategies are applied to align the data size to the desired INPUT_SIZE, including padding, repeating, and pooling the data.

Args:

landmarks (np.ndarray): The input array with landmarks data.

Returns:

Tuple[np.ndarray, int, int, int]: A tuple containing processed landmark data, the set input size, the number of original frames, and the number of frames after preprocessing.

Parameters

`landmarks` (*np.ndarray*) – The input array with landmarks data.

Returns

A tuple containing processed landmark data, the set input size, the number of original frames, and the

number of frames after preprocessing. :rtype: Tuple[np.ndarray, int, int, int]

`data.data_utils.preprocess_raw_data(`*sample=100000*`)`

Preprocesses the raw data, saves it as numpy arrays into processed data directory and updates the metadata CSV file.

This method preprocess_data preprocesses the data for easier and faster loading during training time. The data is processed and stored in PROCESSED_DATA_DIR if not already done.

This function is responsible for preprocessing raw data. The primary functionality involves converting raw data into a format more suitable for the machine learning pipeline, namely NumPy arrays. The function operates on a sample of data, allowing for efficient processing of large datasets in manageable chunks. Additionally, this function also takes care of persisting the preprocessed data for future use and updates the metadata accordingly.

Args: sample (int): Number of samples to preprocess. Default is 100000.

Functionality: - The function reads the metadata CSV file for training data to obtain a dictionary that maps target values to integer indices. - It then reads the training data CSV file and generates the absolute path to locate landmark files. - Next, it keeps text signs and their respective indices and initializes a list to store the processed data. - The data is then processed and stored in the list by iterating over each file path in the training data and reading in the parquet file for that file path. - The landmark data is then processed and padded to have a length of max_seq_length. - Finally, a dictionary with the processed data is created and added to the list. - The processed data is saved to disk using the np.save method and the saved file is printed.

Parameters
`sample` (*int, optional, default: 100000*) – Number of samples to preprocess.

Returns
None

Note: If the preprocessed data already exists, the function prints "Preprocessed data found. Skipping..." and exits.

`data.data_utils.remove_outlier_or_missing_data(`*landmark_len_dict*`)`

This function removes rows from the training data that contain missing or outlier landmark data. It takes as input a dictionary containing the statistics of landmark lengths for each sign type. The function processes the training data and removes rows with missing or outlier landmark data. The function also includes a nested function 'has_consecutive_zeros' which checks for consecutive frames where X and Y coordinates are both zero. If a cleansing marker file exists, it skips the process, indicating that the data is already cleaned.

Functionality:
This function takes a dictionary with the statistics of landmark lengths per sign type and uses it to identify outlier sequences. It removes any rows with missing or outlier landmark data. An outlier sequence is defined as one that is either less than a third of the median length or more than two standard deviations away from the mean length. A row is also marked for deletion if the corresponding landmark file is missing or if the sign's left-hand or right-hand landmarks contain more than a specified number of consecutive zeros.

Args:
landmark_len_dict (dict): A dictionary containing the statistics of landmark lengths for each sign type.

Returns:
None

Parameters
`landmark_len_dict` (*dict*) – A dictionary containing the statistics of landmark lengths for each sign type.

Returns
None, the function doesn't return anything. It modifies data in-place.

`data.data_utils.remove_unusable_data(`*metadata_file='train.csv', skip_check=False*`)`

This function checks the existing training data for unusable instances, like missing files or data that is smaller than the set minimum sequence length. If unusable data is found, it is removed from the system, both in terms of files and entries in the training dataframe. The dataframe is updated and saved back to the disk. If a cleansing marker file exists, it skips the process, indicating that the data is already cleaned.

Functionality:
The function iterates through the DataFrame rows, attempting to load and check each landmark file specified in the row's path. If the file is missing or if the file's usable size is less than a predefined threshold, the function deletes the corresponding landmark file and marks the row for deletion in the

DataFrame. At the end, the function removes all marked rows from the DataFrame, updates it and saves it to the disk.

Returns:

None

Returns

None, the function doesn't return anything. It modifies data in-place.

Chapter 5

HyperParameter Search

This script provides examples for using the MLfowLoggerCallback and setup_mlflow.

It contains functions and classes for defining hyperparameters, training machine learning models, and performing hyperparameter tuning with the aid of MLflow for logging and Ray for distributed computing. The scripts uses ASL (American Sign Language) dataset for training the models.

`class hparam_search.Trainer_HparamSearch(`*modelname='LSTMPredictor', dataset=<class 'data.dataset.ASL_DATASET'>, patience=5*`)`

Trainer_HparamSearch inherits from the Trainer class. It is specifically designed to accommodate hyperparameter search during model training.

Args:
modelname: Name of the model being trained. dataset: Dataset used for training. patience: Number of epochs to wait before stopping the training if no improvement is observed.

Functionality:
Provides a method for model training, taking into account hyperparameter search and early stopping.

`__init__(`*modelname='LSTMPredictor', dataset=<class 'data.dataset.ASL_DATASET'>, patience=5*`)`

Initializes the Trainer class with the specified parameters.

This method initializes various components needed for the training process. This includes the model specified by the model name, the dataset with optional data augmentation and dropout, data loaders for the training, validation, and test sets, a SummaryWriter for logging, and a path for saving model checkpoints.

1. The method first retrieves the specified model and its parameters.
2. It then initializes the dataset and the data loaders.
3. It sets up metrics for early stopping and a writer for logging.
4. Finally, it prepares a directory for saving model checkpoints.

Args:
config (module): the config with the hyperparameters for model training dataset (Dataset): The dataset to be used. model_config (optional):

Functionality:
This method initializes various components, such as the model, dataset, data loaders, logging writer, and checkpoint path, required for the training process.

Param
config: module

Parameters
- `dataset` (*`Dataset or None`*) – The dataset for training.
- `model_config` (*`dict or None`*) – predifined model configuration

Return type
None

Note: This method only initializes the Trainer class. The actual training is done by calling the train() method.

Warning: Make sure the specified model name corresponds to an actual model in your project's models directory.

`train`(*n_epochs=60*)

Train the model for a specific number of epochs. Also implements early stopping based on the patience parameter.

Return type
None

Parameters
`n_epochs` – Number of epochs for training the model.

`hparam_search.train`(*config*)

Function to train a model, with configurable parameters.

Return type
None

Parameters
`config` – Dictionary containing key-value pairs of model configuration parameters.

`hparam_search.tune_with_callback`(*mlflow_tracking_uri*, *finish_fast=False*)

Perform hyperparameter tuning using Ray Tune, with MLflow logging integration.

Return type
None

Parameters
- `mlflow_tracking_uri` – URI for MLflow tracking.
- `finish_fast` – If True, Ray Tune will complete trials as soon as possible.

Chapter 6

Camera Stream Predictions

1 Live Camera Predictions

This script is used to make live sign predictions from a webcam feed or a video file.

Imports: - Required libraries and modules.

`predict_on_camera.show_camera_feed(`*model*, *last_frames=32*, *capture=0*`)`

Function to show live feed from camera or predict a video.

Parameters

- `model` – Pytorch/Tensorflow model for prediction.
- `last_frames` – int, optional The number of frames to use for prediction. Default is INPUT_SIZE.
- `capture` – int or str, optional Choose your webcam (0) or a video file (by entering a path). Default is 0.

Returns

None Displays live feed with prediction results.

Chapter 7

Video Stream Predictions

1 Video Predictions

This script defines methods to predict signs from a given video.

Imports: - Required libraries and modules.

`predict_on_video.get_random_video(`*root_ dir='../data/raw/MSASL/Videos/'*`)`

`predict_on_video.get_top_n_predictions(`*model_ checkpoint*, *landmarks*, *n*`)`

Predicts the top-n signs for given landmarks using the specified model.

Parameters

- `model_checkpoint` – str Path to the model checkpoint.
- `landmarks` – numpy.ndarray Preprocessed video landmarks.
- `n` – int Number of top predictions to return.

Returns

list List of top-n predicted signs.

`predict_on_video.get_video_landmarks(`*video_ path*`)`

Extracts and pre-processes landmarks from a video.

Parameters

`video_path` – str Path to the video file.

Returns

numpy.ndarray Preprocessed video landmarks.

`predict_on_video.play_video_with_predictions(`*video_ path*, *model_ checkpoint*, *num_ top_ predictions*, *sign=''*, *show_ mesh=True*`)`

Plays the video with predicted signs overlay.

Parameters

- `video_path` – str Path to the video file.
- `model_checkpoint` – str Path to the model checkpoint.
- `num_top_predictions` – int Number of top predictions to overlay on the video.
- `sign` – str, optional Name of the sign to display. Default is an empty string.
- `show_mesh` – bool, optional If True, the landmark mesh is drawn on the video. Default is True.

Chapter 8

ASL Dataset

1 ASL Dataset description

This file contains the ASL_DATASET class which serves as the dataset module for American Sign Language (ASL) data. The ASL_DATASET is designed to load, preprocess, augment, and serve the dataset for model training and validation. This class provides functionalities such as loading the dataset from disk, applying transformations, data augmentation techniques, and an interface to access individual data samples.

Note: This dataset class expects data in a specific format. Detailed explanations and expectations about input data are provided in respective method docstrings.

`class data.dataset.ASL_DATASET(`*metadata_df=None*, *transform=None*, *max_seq_length=32*, *augment=False*, *standardize=True*, *augmentation_threshold=0.1*, *load_additional_data=False*, *enableDropout=True*, ***kwargs*`)`

A dataset class for the ASL dataset.

The ASL_DATASET class represents a dataset of American Sign Language (ASL) gestures, where each gesture corresponds to a word or phrase. This class provides functionalities to load the dataset, apply transformations, augment the data, and yield individual data samples for model training and validation.

`__getitem__(`*idx*`)`

Get an item from the dataset by index.

This method returns a data sample from the dataset based on a provided index. It handles reading of the processed data file, applies transformations and augmentations (if set), and pads the data to match the maximum sequence length. It returns the preprocessed landmarks and corresponding target as a tuple.

Args:
idx (int): The index of the item to retrieve.

Returns:
tuple: A tuple containing the landmarks and target for the item.

Functionality:
Get a single item from the dataset.

Parameters
`idx` (*int*) – The index of the item to retrieve.

Returns
A tuple containing the landmarks and target for the item.

Return type
tuple

`__init__`(*metadata_df=None*, *transform=None*, *max_seq_length=32*, *augment=False*, *standardize=True*, *augmentation_threshold=0.1*, *load_additional_data=False*, *enableDropout=True*, ***kwargs*)

Initialize the ASL dataset.

This method initializes the dataset and loads the metadata necessary for the dataset processing. If no metadata is provided, it will load the default processed dataset. It also sets the transformation functions, data augmentation parameters, and maximum sequence length.

Args:
metadata_df (pd.DataFrame, optional): A dataframe containing the metadata for the dataset. Defaults to None. transform (callable, optional): A function/transform to apply to the data. Defaults to None. max_seq_length (int, optional): The maximum sequence length for the data. Defaults to INPUT_SIZE. augment (bool, optional): Whether to apply data augmentation. Defaults to False. augmentation_threshold (float, optional): Probability of augmentation happening. Only if augment == True. Defaults to 0.1. enableDropout (bool, optional): Whether to enable the frame dropout augmentation. Defaults to True.

Functionality:
Initializes the dataset with necessary configurations and loads the data.

Parameters

- `metadata_df` (*pd.DataFrame, optional*) – A dataframe containing the metadata for the dataset.
- `transform` (*callable, optional*) – A function/transform to apply to the data.
- `max_seq_length` (*int*) – The maximum sequence length for the data.
- `augment` (*bool*) – Whether to apply data augmentation.
- `augmentation_threshold` (*float*) – Probability of augmentation happening. Only if augment == True.
- `enableDropout` (*bool*) – Whether to enable the frame dropout augmentation.

`__len__`()

Get the length of the dataset.

This method returns the total number of data samples present in the dataset. It's an implementation of the special method __len__ in Python, providing a way to use the Python built-in function len() on the dataset object.

Functionality:
Get the length of the dataset.

Returns:
int: The length of the dataset.

Returns
The length of the dataset.

Return type
int

`__repr__`()

Return a string representation of the ASL dataset.

This method returns a string that provides an overview of the dataset, including the number of participants and total data samples. It's an implementation of the special method __repr__ in Python, providing a human-readable representation of the dataset object.

Returns:
str: A string representation of the dataset.

Functionality:
Return a string representation of the dataset.

Returns
A string representation of the dataset.

Return type
str

`__weakref__`
list of weak references to the object (if defined)

`load_data(`*load_ additional_ data=False*`)`
Load the data for the ASL dataset.

This method loads the actual ASL data based on the metadata provided during initialization. If no metadata was provided, it loads the default processed data. It generates absolute paths to locate landmark files, and stores individual metadata lists for easy access during data retrieval.

Functionality:
Loads the data for the dataset.

Return type
None

Chapter 9

Data Utilities

1 Deep Learning Utils

This module provides a set of helper functions that abstract away specific details of different deep learning frameworks (such as TensorFlow and PyTorch). These functions allow the main code to run in a framework-agnostic manner, thus improving code portability and flexibility.

`class dl_utils.DatasetWithLen(`*tf_ dataset*, *length*`)`

The DatasetWithLen class serves as a wrapper around TensorFlow's Dataset object. Its primary purpose is to add a length method to the TensorFlow Dataset. This is useful in contexts where it's necessary to know the number of batches that a DataLoader will create from a dataset, which is a common requirement in many machine learning training loops. It also provides an iterator over the dataset, which facilitates traversing the dataset for operations such as batch creation.

For instance, this might be used in conjunction with a progress bar during training to display the total number of batches. Since TensorFlow's Dataset objects don't inherently have a __len__ method, this wrapper class provides that functionality, augmenting the dataset with additional features that facilitate the training process.

Args:
tf_dataset: The TensorFlow dataset to be wrapped. length: The length of the dataset.

Functionality:
Provides a length method and an iterator for a TensorFlow dataset.

Return type
DatasetWithLen object

Parameters
- `tf_dataset` – The TensorFlow dataset to be wrapped.
- `length` – The length of the dataset.

`__init__(`*tf_ dataset*, *length*`)`

`__iter__()`

Returns an iterator for the dataset.

Returns
iterator for the dataset

`__len__()`

Returns the length of the dataset.

Returns
length of the dataset

`__weakref__`

list of weak references to the object (if defined)

`dl_utils.get_PT_Dataset(`*dataloader*`)`

This function retrieves the underlying dataset from a PyTorch DataLoader object.

Functionality:

Extracts the dataset from a PyTorch DataLoader object.

Return type

Dataset

Parameters

`dataloader` – PyTorch DataLoader object.

Returns

The underlying PyTorch Dataset object.

`dl_utils.get_TF_Dataset(`*dataloader*`)`

This function retrieves the underlying dataset from a TensorFlow DataLoader object.

Functionality:

Extracts the dataset from a TensorFlow DataLoader object.

Parameters

`dataloader` – DatasetWithLen object.

Returns

Dataset object.

`dl_utils.get_dataloader(`*dataset*, *batch_ size=128*, *shuffle=True*, *dl_framework='pytorch'*, *num_ workers=16*`)`

The get_dataloader function is responsible for creating a DataLoader object given a dataset and a few other parameters. A DataLoader is an essential component in machine learning projects as it controls how data is fed into the model during training. However, different deep learning frameworks have their own ways of creating and handling DataLoader objects.

To improve the portability and reusability of the code, this function abstracts away these specifics, allowing the user to create a DataLoader object without having to worry about the details of the underlying framework (TensorFlow or PyTorch). This approach can save development time and reduce the risk of bugs or errors.

Args:

dataset: The dataset to be loaded. batch_size: The size of the batches that the DataLoader should create. shuffle: Whether to shuffle the data before creating batches. dl_framework: The name of the deep learning framework. num_workers: The number of worker threads to use for loading data.

Functionality:

Creates and returns a DataLoader object that is compatible with the specified deep learning framework.

Return type

DataLoader or DatasetWithLen object

Parameters

- `dataset` – The dataset to be loaded.
- `batch_size` – The size of the batches that the DataLoader should create.
- `shuffle` – Whether to shuffle the data before creating batches.
- `dl_framework` – The name of the deep learning framework.
- `num_workers` – The number of worker threads to use for loading data.

`dl_utils.get_dataset(`*dataloader*, *dl_framework='pytorch'*`)`

The *get_ dataset* function is an interface to extract the underlying dataset from a dataloader, irrespective of the deep learning framework being used, i.e., TensorFlow or PyTorch. The versatility of this function makes it integral to any pipeline designed to be flexible across both TensorFlow and PyTorch frameworks.

Given a dataloader object, this function first determines the deep learning framework currently in use by referring to the *DL_FRAMEWORK* config parameter variable. If the framework is TensorFlow, it invokes the *get_TF_Dataset* function to retrieve the dataset. Alternatively, if PyTorch is being used, the *get_PT_Dataset* function is called. This abstracts away the intricacies of handling different deep learning frameworks, thereby simplifying the process of working with datasets across TensorFlow and PyTorch.

Args:
dataloader: DataLoader from PyTorch or DatasetWithLen from TensorFlow.

Functionality:
Extracts the underlying dataset from a dataloader, be it from PyTorch or TensorFlow.

Return type
Dataset object

Parameters
`dataloader` – DataLoader in case of PyTorch and DatasetWithLen in case of TensorFlow.

`dl_utils.get_metric_dict()`

This function is responsible for creating an empty dictionary, structured to log the specified metrics for different phases (Train, Validation, Test) in Tensorboard. The function initializes an empty list for each metric and phase combination, and these lists are then mapped to their corresponding keys (metric/phase) in the dictionary.

Functionality:
Creates and returns a dictionary with None as initial values for the metric and phase keys.

Return type
dict

Returns
Dictionary with keys for each metric and phase, all initialized to None.

`dl_utils.get_model_params(`*model_name*`)`

The get_model_params function is a utility function that serves to abstract away the details of reading model configurations from a YAML file. In a machine learning project, it is common to have numerous models, each with its own set of hyperparameters. These hyperparameters can be stored in a YAML file for easy access and modification.

This function reads the configuration file and retrieves the specific parameters associated with the given model. The configurations are stored in a dictionary which is then returned. This aids in maintaining a cleaner, more organized codebase and simplifies the process of updating or modifying model parameters.

Args:
model_name: Name of the model whose parameters are to be retrieved.

Functionality:
Reads a YAML file and retrieves the model parameters as a dictionary.

Return type
dict

Parameters
`model_name` – Name of the model whose parameters are to be retrieved.

`dl_utils.load_model_from_checkpoint(`*ckpt_name*`)`

This function loads a deep learning model from a previously saved checkpoint. It is useful when you want to resume training from a certain point or when you want to use a pre-trained model. This function takes the name of the checkpoint as input and returns the model in the device specified by the DEVICE global variable.

The function first constructs the paths to the checkpoint and YAML files containing model parameters. It then reads the YAML file and extracts the model parameters. Using importlib, the function dynamically imports the correct model class based on the model name extracted from the checkpoint name and the deep learning framework specified in the DL_FRAMEWORK global variable. The model is instantiated with the extracted parameters, loaded from the checkpoint, and moved to the appropriate device.

Args:
ckpt_name: The name of the checkpoint from which the model should be loaded.

Functionality:
Loads a model from a checkpoint file and moves it to a specified device.

Return type
Model

Parameters
`ckpt_name` – The name of the checkpoint from which the model should be loaded.

Returns
The model loaded from the checkpoint, moved to the specified device.

`dl_utils.log_hparams_metrics(`*writer*, *hparam_dict*, *metric_dict*, *epoch=0*`)`

Helper function to log metrics to TensorBoard. That accepts the logging of hyperparameters too. It allows to display the hyperparameters as well in a tensorboard instance. Furthermore it logs everything in just one tensorboard log.

Parameters
- `writer` (*torch.utils.tensorboard.SummaryWriter*) – Summary Writer Object
- `hparam_dict` (*dict*) –
- `metric_dict` (*dict*) –
- `epoch` (*int*) – Step on the x-axis to log the results

`dl_utils.log_metrics(`*writer*, *log_dict*`)`

Helper function to log metrics to TensorBoard.

Parameters
- `log_dict` – Dictionary to log all the metrics to tensorboard. It must contain the keys {epoch,accuracy, loss, lr,}
- `writer` – TensorBoard writer object.

Type
log_dict

`dl_utils.to_PT_DataLoader(`*dataset*, *batch_size=128*, *shuffle=True*, *num_workers=16*`)`

This function is the PyTorch counterpart to 'to_TF_DataLoader'. It converts a given dataset into a PyTorch DataLoader. The purpose of this function is to streamline the creation of PyTorch DataLoaders, allowing for easy utilization in a PyTorch training or inference pipeline.

The PyTorch DataLoader handles the process of drawing batches of data from a dataset, which is essential when training models. This function further extends this functionality by implementing data shuffling and utilizing multiple worker threads for asynchronous data loading, thereby optimizing the data loading process during model training.

Args:
dataset: The dataset to be loaded. batch_size: The size of each batch the DataLoader will return. shuffle: Whether the data should be shuffled before batching. num_workers: The number of worker threads to use for data loading.

Functionality:
Converts a given dataset into a PyTorch DataLoader.

Return type
DataLoader object

Parameters
- `dataset` – The dataset to be loaded.
- `batch_size` – The size of each batch the DataLoader will return.
- `shuffle` – Whether the data should be shuffled before batching.
- `num_workers` – The number of worker threads to use for data loading.

`dl_utils.to_TF_DataLoader(`*dataset*, *batch_size=128*, *shuffle=True*`)`

This function takes in a dataset and converts it into a TensorFlow DataLoader. Its purpose is to provide a streamlined method to generate DataLoaders that can be utilized in a TensorFlow training or inference pipeline. It not only ensures the dataset is in a format that can be ingested by TensorFlow's pipeline, but also implements optional shuffling of data, which is a common practice in model training to ensure random distribution of data across batches.

This function first checks whether the data is already in a tensor format, if not it converts the data to a tensor. Next, it either shuffles the dataset or keeps it as is, based on the 'shuffle' flag. Lastly, it prepares the TensorFlow DataLoader by batching the dataset and applying an automatic optimization strategy for the number of parallel calls in mapping functions.

Args:

dataset: The dataset to be loaded. batch_size: The size of each batch the DataLoader will return. shuffle: Whether the data should be shuffled before batching.

Functionality:

Converts a given dataset into a TensorFlow DataLoader.

Return type

DatasetWithLen object

Parameters

- `dataset` – The dataset to be loaded.
- `batch_size` – The size of each batch the DataLoader will return.
- `shuffle` – Whether the data should be shuffled before batching.

Chapter 10

Model Training

1 Generic trainer description

Trainer module handles the training, validation, and testing of framework-agnostic deep learning models.

The Trainer class handles the complete lifecycle of model training including setup, execution of training epochs, validation and testing, early stopping, and result logging.

The class uses configurable parameters for defining training settings like early stopping and batch size, and it supports adding custom callback functions to be executed at the end of each epoch. This makes the trainer class flexible and adaptable for various types of deep learning models and tasks.

Attributes: model_name (str): The name of the model to be trained. params (dict): The parameters required for the model. model (model object): The model object built using the given model name and parameters. train_loader, valid_loader, test_loader (DataLoader objects): PyTorch dataloaders for training, validation, and testing datasets. patience (int): The number of epochs to wait before stopping training when the validation loss is no longer improving. best_val_metric (float): The best validation metric recorded. patience_counter (int): A counter that keeps track of the number of epochs since the validation loss last improved. model_class (str): The class name of the model. train_start_time (str): The starting time of the training process. writer (SummaryWriter object): TensorBoard's SummaryWriter to log metrics for visualization. checkpoint_path (str): The path where the best model checkpoints will be saved during training. epoch (int): The current epoch number. callbacks (list): A list of callback functions to be called at the end of each epoch.

Methods: train(n_epochs): Trains the model for a specified number of epochs. evaluate(): Evaluates the model on the validation set. test(): Tests the model on the test set. add_callback(callback): Adds a callback function to the list of functions to be called at the end of each epoch.

`class trainer.Trainer(`*config*, *dataset=<class 'data.dataset.ASL_DATASET'>*, *model_config=None*, *Model_Name=None*`)`

A trainer class which acts as a control hub for the model lifecycle, including initial setup, executing training epochs, performing validation and testing, implementing early stopping, and logging results. The module has been designed to be agnostic to the specific deep learning framework, enhancing its versatility across various projects.

`__init__(`*config*, *dataset=<class 'data.dataset.ASL_DATASET'>*, *model_config=None*, *Model_Name=None*`)`

Initializes the Trainer class with the specified parameters.

This method initializes various components needed for the training process. This includes the model specified by the model name, the dataset with optional data augmentation and dropout, data loaders for the training, validation, and test sets, a SummaryWriter for logging, and a path for saving model checkpoints.

1. The method first retrieves the specified model and its parameters.
2. It then initializes the dataset and the data loaders.

3. It sets up metrics for early stopping and a writer for logging.
4. Finally, it prepares a directory for saving model checkpoints.

Args:

config (module): the config with the hyperparameters for model training dataset (Dataset): The dataset to be used. model_config (optional):

Functionality:

This method initializes various components, such as the model, dataset, data loaders, logging writer, and checkpoint path, required for the training process.

Param

config: module

Parameters

- `dataset` (*Dataset or None*) – The dataset for training.
- `model_config` (*dict or None*) – predifined model configuration

Return type

None

Note: This method only initializes the Trainer class. The actual training is done by calling the train() method.

Warning: Make sure the specified model name corresponds to an actual model in your project's models directory.

`__weakref__`

list of weak references to the object (if defined)

`add_callback(`*callback*`)`

Adds a callback to the Trainer.

This method simply appends a callback function to the list of callbacks stored by the Trainer instance. These callbacks are called at the end of each training epoch.

Functionality:

It allows the addition of custom callbacks to the training process, enhancing its flexibility.

Parameters

`callback` (*Callable*) – The callback function to be added.

Returns

None

Return type

None

Warning: The callback function must be callable and should not modify the training process.

`evaluate()`

Evaluates the model on the validation set.

This method sets the model to evaluation mode and loops over the validation dataset, computing the loss and accuracy for each batch. It then averages these metrics and logs them. This process provides an unbiased estimate of the model's performance on new data during training.

Functionality:

It manages the evaluation of the model on the validation set, handling batch-wise loss computation and accuracy assessment.

Returns
Average validation loss and accuracy

Return type
Tuple[float, float]

> **Warning:** Ensure the model is in evaluation mode to correctly compute the validation metrics.

`get_classification_report`(*labels*, *preds*, *verbose=True*)

Generates Classification report :param labels: tensor with True values :param preds: tensor with predicted values :param verbose: Wherer to print the report :type: verbose: bool :return: classification report

`test`()

Tests the model on the test set.

This method loads the best saved model, sets it to evaluation mode, and then loops over the test dataset, computing the loss, accuracy, and predictions for each batch. It then averages the loss and accuracy and logs them. It also collects all the model's predictions and their corresponding labels.

Functionality:
It manages the testing of the model on the test set, handling batch-wise loss computation, accuracy assessment, and prediction generation.

Returns
List of all predictions and their corresponding labels

Return type
Tuple[List, List]

`train`(*n_ epochs=60*)

Trains the model for a specified number of epochs.

This method manages the main training loop of the model. For each epoch, it performs several steps. It first puts the model into training mode and loops over the training dataset, calculating the loss and accuracy for each batch and optimizing the model parameters. It logs these metrics and updates a progress bar. At the end of each epoch, it evaluates the model on the validation set and checks whether early stopping criteria have been met. If the early stopping metric has improved, it saves the current model and its parameters. If not, it increments a counter and potentially stops training if the counter exceeds the allowed patience. Finally, it steps the learning rate scheduler and calls any registered callbacks.

1. The method first puts the model into training mode and initializes some lists and counters.
2. Then it enters the main loop over the training data, updating the model and logging metrics.
3. It evaluates the model on the validation set and checks the early stopping criteria.
4. If the criteria are met, it saves the model and its parameters; if not, it increments a patience counter.
5. It steps the learning rate scheduler and calls any callbacks.

Args:
n_epochs (int): The number of epochs for which the model should be trained.

Functionality:
This method coordinates the training of the model over a series of epochs, handling batch-wise loss computation, backpropagation, optimization, validation, early stopping, and model checkpoint saving.

Parameters
`n_epochs` (*int*) – Number of epochs for training.

Returns
None

Return type
None

Note: This method modifies the state of the model and its optimizer, as well as various attributes of the Trainer instance itself.

Warning: If you set the patience value too low in the constructor, the model might stop training prematurely.

`write_classification_report`(*preds*, *labels*)

Function to export classification report

Args:

Parameters

- `preds` (*tensor*) – tensor
- `labels` – tensor

Returns

None

Chapter 11

Torch Lightning Models

class models.pytorch.lightning_models.LightningBaseModel(**args: Any, **kwargs: Any*)

`__init__`(*learning_rate, n_classes=250*)

`configure_optimizers()`

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple. Optimization with multiple optimizers only works in the manual optimization mode.

Return:

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
```

(continues on next page)

(continued from previous page)

```
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note:

Some things to know:

- Lightning calls `.backward()` and `.step()` automatically in case of automatic optimization.
- If a learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizer.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, you will have to switch to 'manual optimization' mode and step them yourself.
- If you need to control how often the optimizer steps, override the `optimizer_step()` hook.

`forward(`*x*`)`

Same as `torch.nn.Module.forward()`.

Args:

*args: Whatever you decide to pass into the forward method. **kwargs: Keyword arguments are also possible.

Return:

Your model's output

`on_test_end()` → None

Called at the end of testing.

`on_train_epoch_end()` → None

Called in the training loop at the very end of the epoch.

To access all batch outputs at the end of the epoch, you can cache step outputs as an attribute of the `LightningModule` and access them in this hook:

```python
class MyLightningModule(L.LightningModule):
    def __init__(self):
        super().__init__()
        self.training_step_outputs = []

    def training_step(self):
        loss = ...
        self.training_step_outputs.append(loss)
        return loss

    def on_train_epoch_end(self):
        # do something with all training_step outputs, for example:
        epoch_mean = torch.stack(self.training_step_outputs).mean()
        self.log("training_epoch_mean", epoch_mean)
        # free up the memory
        self.training_step_outputs.clear()
```

`on_validation_end()`

Called at the end of validation.

`test_step`(*batch*, *batch_idx*)

Operates on a single batch of data from the test set. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

Args:

batch: The output of your `DataLoader`. batch_idx: The index of this batch. dataloader_id: The index of the dataloader that produced this batch.
(only if multiple test dataloaders used).

Return:

Any of.

- Any object or value
- `None` - Testing will skip to the next batch

```
# if you have one test dataloader:
def test_step(self, batch, batch_idx):
    ...


# if you have multiple test dataloaders:
def test_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single test dataset
def test_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    test_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'test_loss': loss, 'test_acc': test_acc})
```

If you pass in multiple test dataloaders, *`test_step()`* will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple test dataloaders
def test_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note:

If you don't need to test you don't need to implement this method.

Note:

When the *`test_step()`* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of the test epoch, the model goes back to training mode and gradients are enabled.

`training_step`(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Args:

batch (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]):
The output of your `DataLoader`. A tensor, tuple or list.

batch_idx (`int`): Integer displaying index of this batch

Return:

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- `None` - **Training will skip to the next batch. This is only for automatic optimization.**
  This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```python
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

To use multiple optimizers, you can switch to 'manual optimization' and control their stepping:

```python
def __init__(self):
    super().__init__()
    self.automatic_optimization = False


# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx):
    opt1, opt2 = self.optimizers()

    # do training_step with encoder
    ...
    opt1.step()
    # do training_step with decoder
    ...
    opt2.step()
```

Note:

When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

`validation_step`(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

Args:

batch: The output of your `DataLoader`. batch_idx: The index of this batch. dataloader_idx: The index of the dataloader that produced this batch.
(only if multiple val dataloaders used)

Return:

- Any object or value
- `None` - Validation will skip to the next batch

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, *validation_step()* will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note:
If you don't need to validate you don't need to implement this method.

Note:
When the *validation_step()* is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

`class models.pytorch.lightning_models.LightningTransformerPredictor(`**args: Any*, ***kwargs: Any*`)`

`__init__(`***kwargs*`)`

`forward(`*x*`)`

Same as `torch.nn.Module.forward()`.

Args:
*args: Whatever you decide to pass into the forward method. **kwargs: Keyword arguments are also possible.

Return:
Your model's output

`class models.pytorch.lightning_models.LightningTransformerSequenceClassifier(`**args: Any*, ***kwargs: Any*`)`

Transformer-based Sequence Classifier

`__init__(`***kwargs*`)`

`forward(`*inputs*`)`

Forward pass through the model

Chapter 12

Pytorch Models

This module defines a PyTorch *BaseModel* providing a basic framework for learning and validating from *Trainer* module, from which other pytorch models are inherited. This module includes several model classes that build upon the PyTorch's nn.Module for constructing pytorch LSTM or Transformer based models:

Table 1: Model Classes

Class	Description
TransformerSequence-Classifier	This is a transformer-based sequence classification model. The class constructs a transformer encoder based on user-defined parameters or default settings. The forward method first checks and reshapes the input, then passes it through the transformer layers. It then pools the sequence by taking the mean over the time dimension, and finally applies the output layer to generate the class predictions.
TransformerPredictor	A TransformerPredictor model that extends the Pytorch BaseModel. This class wraps *TransformerSequenceClassifier* model and provides functionality to use it for making predictions.
MultiHeadSelfAttention	This class applies a multi-head attention mechanism. It has options for causal masking and layer normalization. The input is expected to have dimensions [batch_size, seq_len, features].
TransformerBlock	This class represents a single block of a transformer architecture, including multi-head self-attention and a feed-forward neural network, both with optional layer normalization and dropout. The input is expected to have dimensions [batch_size, seq_len, features].
YetAnotherTransformer-Classifier	This class constructs a transformer-based classifier with a specified number of *TransformerBlock* instances. The output of the model is a tensor of logits with dimensions [batch_size, num_classes].
YetAnotherTransformer	This class is a wrapper for *YetAnotherTransformerClassifier* which includes learning rate, optimizer, and learning rate scheduler settings. It extends from the *BaseModel* class.
YetAnotherEnsemble	This class constructs an ensemble of *YetAnotherTransformerClassifier* instances, where the outputs are concatenated and passed through a fully connected layer. This class also extends from the *BaseModel* class and includes learning rate, optimizer, and learning rate scheduler settings.

`class models.pytorch.models.BaseModel(`**args: Any*, ***kwargs: Any*`)`

A BaseModel that extends the nn.Module from PyTorch.

Functionality: #. The class initializes with a given learning rate and number of classes. #. It sets up the loss criterion, accuracy metric, and default states for optimizer and scheduler. #. It defines an abstract method 'forward' which should be implemented in the subclass. #. It also defines various utility functions like calculating accuracy, training, validation and testing steps, scheduler stepping, and model checkpointing.

Args:

learning_rate (float): The initial learning rate for optimizer. n_classes (int): The number of classes for classification.

Parameters
- `learning_rate` (*float*) – The initial learning rate for optimizer.
- `n_classes` (*int*) – The number of classes for classification.

Returns
None

Return type
None

Note: The class does not directly initialize the optimizer and scheduler. They should be initialized in the subclass if needed.

Warning: The 'forward' function must be implemented in the subclass, else it will raise a NotImplementedError.

`__init__`(*learning_rate*, *n_classes=250*)

`calculate_accuracy`(*y_hat*, *y*)

Calculates the accuracy of the model's prediction.

Parameters
- `y_hat` (*Tensor*) – The predicted output from the model.
- `y` (*Tensor*) – The ground truth or actual labels.

Returns
The calculated accuracy.

Return type
Tensor

`calculate_auc`(*y_hat*, *y*)

Calculates the auc of the model's prediction.

Parameters
- `y_hat` (*Tensor*) – The predicted output from the model.
- `y` (*Tensor*) – The ground truth or actual labels.

Returns
The calculated recall.

Return type
Tensor

`calculate_f1score`(*y_hat*, *y*)

Calculates the F1-Score of the model's prediction.

Parameters
- `y_hat` (*Tensor*) – The predicted output from the model.
- `y` (*Tensor*) – The ground truth or actual labels.

Returns
The calculated f1.

Return type
Tensor

`calculate_precision`(*y_hat*, *y*)

Calculates the precision of the model's prediction.

Parameters
- `y_hat` (*Tensor*) – The predicted output from the model.
- `y` (*Tensor*) – The ground truth or actual labels.

Returns
The calculated precision.

Return type
Tensor

`calculate_recall(`*y_hat*, *y*`)`

Calculates the recall of the model's prediction.

Parameters
- `y_hat` (*Tensor*) – The predicted output from the model.
- `y` (*Tensor*) – The ground truth or actual labels.

Returns
The calculated recall.

Return type
Tensor

`eval_mode()`

Sets the model to evaluation mode.

`forward(`*x*`)`

The forward function for the BaseModel.

Parameters
`x` (*Tensor*) – The inputs to the model.

Returns
None

Warning: This function must be implemented in the subclass, else it raises a NotImplementedError.

`get_lr()`

Gets the current learning rate of the model.

Returns
The current learning rate.

Return type
float

`load_checkpoint(`*filepath*`)`

Loads the model and optimizer states from a checkpoint.

Parameters
`filepath` (*str*) – The file path where to load the model checkpoint from.

`optimize()`

Steps the optimizer and sets the gradients of all optimized `torch.Tensor` s to zero.

`save_checkpoint(`*filepath*`)`

Saves the model and optimizer states to a checkpoint.

Parameters
`filepath` (*str*) – The file path where to save the model checkpoint.

`step_scheduler()`

Steps the learning rate scheduler, adjusting the optimizer's learning rate as necessary.

`test_step(`*batch*`)`

Performs a test step using the input batch data.

Parameters
`batch` (*tuple*) – A tuple containing input data and labels.

Returns
The calculated loss, accuracy, labels, and model predictions.

Return type
tuple

`train_mode()`

Sets the model to training mode.

`training_step(`*batch*`)`

Performs a training step using the input batch data.

Parameters
`batch` (*tuple*) – A tuple containing input data and labels.
Returns
The calculated loss and accuracy, labels and predictions
Return type
tuple

`validation_step(`*batch*`)`

Performs a validation step using the input batch data.

Parameters
`batch` (*tuple*) – A tuple containing input data and labels.
Returns
The calculated loss and accuracy, labels and predictions
Return type
tuple

`class models.pytorch.models.CVTransferLearningModel(`**args: Any*, ***kwargs: Any*`)`

1 CVTransferLearningModel

A CVTransferLearningModel that extends the Pytorch BaseModel.

This class applies transfer learning for computer vision tasks using pretrained models. It also provides a forward method to pass an input through the model.

1.1 Attributes

learning_rate
[float] The learning rate for the optimizer.
model
[nn.Module] The base model for transfer learning.
optimizer
[torch.optim.Adam] The optimizer used for updating the model parameters.
scheduler
[torch.optim.lr_scheduler.ExponentialLR] The learning rate scheduler used for adapting the learning rate during training.

1.2 Methods

forward(x)
Performs a forward pass through the model.

`__init__(`***kwargs*`)`

`forward(`*x*`)`

The forward function for the BaseModel.

Parameters
`x` (*Tensor*) – The inputs to the model.

Returns
None

Warning: This function must be implemented in the subclass, else it raises a NotImplementedError.

`class models.pytorch.models.HybridEnsembleModel(`**args: Any*, ***kwargs: Any*`)`

2 HybridEnsembleModel

A HybridEnsembleModel that extends the Pytorch BaseModel.

This class creates an ensemble of LSTM and Transformer models and provides functionality to use the ensemble for making predictions.

2.1 Attributes

learning_rate
[float] The learning rate for the optimizer.

lstms
[nn.ModuleList] The list of LSTM models.

models
[nn.ModuleList] The list of Transformer models.

fc
[nn.Linear] The final fully-connected layer.

optimizer
[torch.optim.Adam] The optimizer used for updating the model parameters.

scheduler
[torch.optim.lr_scheduler.ExponentialLR] The learning rate scheduler used for adapting the learning rate during training.

2.2 Methods

forward(x)
Performs a forward pass through the model.

`__init__(`***kwargs*`)`

`forward(`*x*`)`

The forward function for the BaseModel.

Parameters
`x` (*Tensor*) – The inputs to the model.

Returns
None

> **Warning:** This function must be implemented in the subclass, else it raises a NotImplementedError.

`class models.pytorch.models.HybridModel(`**args: Any*, ***kwargs: Any*`)`

3 HybridModel

A HybridModel that extends the Pytorch BaseModel.

This class combines the LSTMClassifier and TransformerSequenceClassifier models and provides functionality to use the combined model for making predictions.

3.1 Attributes

lstm
[LSTMClassifier] The LSTM classifier used for making predictions.

transformer
[TransformerSequenceClassifier] The transformer sequence classifier used for making predictions.

fc
[nn.Linear] The final fully-connected layer.

optimizer
[torch.optim.Adam] The optimizer used for updating the model parameters.

scheduler
[torch.optim.lr_scheduler.ExponentialLR] The learning rate scheduler used for adapting the learning rate during training.

3.2 Methods

forward(x)
Performs a forward pass through the model.

`__init__(`***kwargs*`)`

`forward(`*x*`)`

The forward function for the BaseModel.

Parameters
x (*Tensor*) – The inputs to the model.

Returns
None

> **Warning:** This function must be implemented in the subclass, else it raises a NotImplementedError.

`class models.pytorch.models.LSTMClassifier(`**args: Any*, ***kwargs: Any*`)`

4 LSTMClassifier

A LSTM-based Sequence Classifier. This class utilizes a LSTM network for sequence classification tasks.

4.1 Attributes

DEFAULTS
[dict] Default settings for the LSTM and classifier. These can be overridden by passing values in the constructor.

lstm
[nn.LSTM] The LSTM network used for processing the input sequence.

dropout
[nn.Dropout] The dropout layer applied after LSTM network.

output_layer
[nn.Linear] The output layer used to generate class predictions.

4.2 Methods

forward(x)
Performs a forward pass through the model.

`__init__`(***kwargs*)

`forward`(*x*)

Forward pass through the model

`class models.pytorch.models.LSTMPredictor`(**args: Any*, ***kwargs: Any*)

5 LSTMPredictor

A LSTMPredictor model that extends the Pytorch BaseModel.

This class wraps the LSTMClassifier model and provides functionality to use it for making predictions.

5.1 Attributes

learning_rate
[float] The learning rate for the optimizer.

model
[LSTMClassifier] The LSTM classifier used for making predictions.

optimizer
[torch.optim.Adam] The optimizer used for updating the model parameters.

scheduler
[torch.optim.lr_scheduler.ExponentialLR] The learning rate scheduler used for adapting the learning rate during training.

5.2 Methods

forward(x)
Performs a forward pass through the model.

`__init__`(***kwargs*)

`forward`(*x*)

The forward function for the BaseModel.

Parameters
x (*Tensor*) – The inputs to the model.

Returns
None

> **Warning:** This function must be implemented in the subclass, else it raises a NotImplementedError.

`class models.pytorch.models.MultiHeadSelfAttention`(**args: Any*, ***kwargs: Any*)

6 MultiHeadSelfAttention

A MultiHeadSelfAttention module that extends the nn.Module from PyTorch.

Functionality: #. The class initializes with a given dimension size, number of attention heads, dropout rate, layer normalization and causality. #. It sets up the multihead attention module and layer normalization. #. It also defines a forward method that applies the multihead attention, causal masking if requested, and layer normalization if requested.

6.1 Attributes

multihead_attn
[nn.MultiheadAttention] The multihead attention module.

layer_norm
[nn.LayerNorm or None] The layer normalization module. If it is not applied, set to None.

causal
[bool] If True, applies causal masking.

6.2 Methods

forward(x)
Performs a forward pass through the model.

Args:
dim (int): The dimension size of the input data. num_heads (int): The number of attention heads. dropout (float): The dropout rate. layer_norm (bool): Whether to apply layer normalization. causal (bool): Whether to apply causal masking.

Returns: None

`__init__`(*dim*, *num_heads=8*, *dropout=0.1*, *layer_norm=True*, *causal=True*)

`class models.pytorch.models.TransformerBlock`(**args: Any*, ***kwargs: Any*)

7 TransformerBlock

A TransformerBlock module that extends the nn.Module from PyTorch.

Functionality: #. The class initializes with a given dimension size, number of attention heads, expansion factor, attention dropout rate, and dropout rate. #. It sets up the multihead self-attention module, layer normalization and feed-forward network. #. It also defines a forward method that applies the multihead self-attention, dropout, layer normalization and feed-forward network.

7.1 Attributes

norm1, norm2, norm3
[nn.LayerNorm] The layer normalization modules.

attn
[MultiHeadSelfAttention] The multihead self-attention module.

feed_forward
[nn.Sequential] The feed-forward network.

dropout
[nn.Dropout] The dropout module.

7.2 Methods

forward(x)
Performs a forward pass through the model.

Args:
dim (int): The dimension size of the input data. num_heads (int): The number of attention heads. expansion_factor (int): The expansion factor for the hidden layer size in the feed-forward network. attn_dropout (float): The dropout rate for the attention module. drop_rate (float): The dropout rate for the module.

Returns: None

`__init__`(*dim=192*, *num_heads=4*, *expansion_factor=4*, *attn_dropout=0.2*, *drop_rate=0.2*)

`class models.pytorch.models.TransformerEnsemble`(**args: Any*, ***kwargs: Any*)

8 TransformerEnsemble

A TransformerEnsemble that extends the Pytorch BaseModel.

This class creates an ensemble of TransformerSequenceClassifier models and provides functionality to use the ensemble for making predictions.

8.1 Attributes

learning_rate
[float] The learning rate for the optimizer.

models
[nn.ModuleList] The list of transformer sequence classifiers used for making predictions.

fc
[nn.Linear] The final fully-connected layer.

optimizer
[torch.optim.Adam] The optimizer used for updating the model parameters.

scheduler
: [torch.optim.lr_scheduler.ExponentialLR] The learning rate scheduler used for adapting the learning rate during training.

8.2 Methods

forward(x)
: Performs a forward pass through the model.

`__init__`(***kwargs*)

`forward`(*x*)

The forward function for the BaseModel.

Parameters
: x (*Tensor*) – The inputs to the model.

Returns
: None

Warning: This function must be implemented in the subclass, else it raises a NotImplementedError.

`class models.pytorch.models.TransformerPredictor`(**args: Any*, ***kwargs: Any*)

9 TransformerPredictor

A TransformerPredictor model that extends the Pytorch BaseModel.

This class wraps the TransformerSequenceClassifier model and provides functionality to use it for making predictions.

9.1 Attributes

learning_rate
: [float] The learning rate for the optimizer.

model
: [TransformerSequenceClassifier] The transformer sequence classifier used for making predictions.

optimizer
: [torch.optim.Adam] The optimizer used for updating the model parameters.

scheduler
: [torch.optim.lr_scheduler.ExponentialLR] The learning rate scheduler used for adapting the learning rate during training.

9.2 Methods

forward(x)
: Performs a forward pass through the model.

`__init__`(***kwargs*)

`forward`(*x*)

The forward function for the BaseModel.

Parameters
: x (*Tensor*) – The inputs to the model.

Returns
None

Warning: This function must be implemented in the subclass, else it raises a NotImplementedError.

`class models.pytorch.models.TransformerSequenceClassifier(`**args: Any*, ***kwargs: Any*`)`

10 TransformerSequenceClassifier

A Transformer-based Sequence Classifier. This class utilizes a transformer encoder to process the input sequence.

The transformer encoder consists of a stack of N transformer layers that are applied to the input sequence. The output sequence from the transformer encoder is then passed through a linear layer to generate class predictions.

10.1 Attributes

DEFAULTS
[dict] Default settings for the transformer encoder and classifier. These can be overridden by passing values in the constructor.

transformer
[nn.TransformerEncoder] The transformer encoder used to process the input sequence.

output_layer
[nn.Linear] The output layer used to generate class predictions.

batch_first
[bool] Whether the first dimension of the input tensor represents the batch size.

10.2 Methods

forward(inputs)
Performs a forward pass through the model.

`__init__(`***kwargs*`)`

`forward(`*inputs*`)`

Forward pass through the model

`class models.pytorch.models.YetAnotherEnsemble(`**args: Any*, ***kwargs: Any*`)`

11 YetAnotherEnsemble

A YetAnotherEnsemble model that extends the Pytorch BaseModel.

Functionality: #. The class initializes with a set of parameters for the YetAnotherTransformerClassifier. #. It sets up an ensemble of YetAnotherTransformerClassifier models, a fully connected layer, the optimizer and the learning rate scheduler. #. It also defines a forward method that applies each YetAnotherTransformerClassifier model in the ensemble, concatenates the outputs and applies the fully connected layer.

Args:
kwargs (dict): A dictionary containing the parameters for the YetAnotherTransformerClassifier models, fully connected layer, optimizer and learning rate scheduler.

Returns: None

`__init__`(***kwargs*)

`forward`(*x*)

The forward function for the BaseModel.

Parameters
x (*Tensor*) – The inputs to the model.

Returns
None

Warning: This function must be implemented in the subclass, else it raises a NotImplementedError.

`class models.pytorch.models.YetAnotherTransformer`(**args: Any*, ***kwargs: Any*)

12 YetAnotherTransformer

A YetAnotherTransformer model that extends the Pytorch BaseModel.

Functionality: #. The class initializes with a set of parameters for the YetAnotherTransformerClassifier. #. It sets up the YetAnotherTransformerClassifier model, the optimizer and the learning rate scheduler. #. It also defines a forward method that applies the YetAnotherTransformerClassifier model.

12.1 Attributes

learning_rate
[float] The learning rate for the optimizer.

model
[YetAnotherTransformerClassifier] The YetAnotherTransformerClassifier model.

optimizer
[torch.optim.AdamW] The AdamW optimizer.

scheduler
[torch.optim.lr_scheduler.ExponentialLR] The learning rate scheduler.

12.2 Methods

forward(x)
Performs a forward pass through the model.

Args:
kwargs (dict): A dictionary containing the parameters for the YetAnotherTransformerClassifier, optimizer and learning rate scheduler.

Returns: None

`__init__`(***kwargs*)

`forward`(*x*)

The forward function for the BaseModel.

Parameters
x (*Tensor*) – The inputs to the model.

Returns
None

Warning: This function must be implemented in the subclass, else it raises a NotImplementedError.

`class models.pytorch.models.YetAnotherTransformerClassifier(`*`*args: Any, **kwargs: Any`*`)`

13 YetAnotherTransformerClassifier

A YetAnotherTransformerClassifier module that extends the nn.Module from PyTorch.

Functionality: #. The class initializes with a set of parameters for the transformer blocks. #. It sets up the transformer blocks and the output layer. #. It also defines a forward method that applies the transformer blocks, takes the mean over the time dimension of the transformed sequence, and applies the output layer.

13.1 Attributes

DEFAULTS
: [dict] The default settings for the transformer.

settings
: [dict] The settings for the transformer, with any user-provided values overriding the defaults.

transformer
: [nn.ModuleList] The list of transformer blocks.

output_layer
: [nn.Linear] The output layer.

13.2 Methods

forward(inputs)
: Performs a forward pass through the model.

Args:
: kwargs (dict): A dictionary containing the parameters for the transformer blocks.

Returns: None

`__init__(`*`**kwargs`*`)`

`forward(`*`inputs`*`)`
: Forward pass through the model

Chapter 13

Tensorflow Models

This kodule defines a PyTorch *BaseModel* providing a basic framework for learning and validating from *Trainer*

`class models.tensorflow.models.BaseModel(`**args: Any*, ***kwargs: Any*`)`

A BaseModel that extends the tf.keras.Model.

Functionality: #. The class initializes with a given learning rate. #. It sets up the loss criterion, accuracy metric, and default states for optimizer and scheduler. #. It defines an abstract method 'call' which should be implemented in the subclass. #. It also defines various utility functions like calculating accuracy, training, validation and testing steps, scheduler stepping, and model checkpointing.

Args:
learning_rate (float): The initial learning rate for optimizer.

Parameters
`learning_rate` (*float*) – The initial learning rate for optimizer.

Returns
None

Return type
None

Note: The class does not directly initialize the optimizer and scheduler. They should be initialized in the subclass if needed.

Warning: The 'call' function must be implemented in the subclass, else it will raise a NotImplementedError.

`__init__(`*learning_rate*`)`

`calculate_accuracy(`*y_pred*, *y_true*`)`

Calculates the accuracy of the model's prediction.

Parameters
- `y_pred` (*Tensor*) – The predicted output from the model.
- `y_true` (*Tensor*) – The ground truth or actual labels.

Returns
The calculated accuracy.

Return type
float

`calculate_auc(`*y_pred*, *y_true*`)`

Calculates the AUC of the model's prediction.

Parameters
- `y_pred` (*Tensor*) – The predicted output from the model.
- `y_true` (*Tensor*) – The ground truth or actual labels.

Returns
The calculated accuracy.

Return type
float

`calculate_f1score(`*y_pred*, *y_true*`)`

Calculates the F1-Score of the model's prediction.

Parameters
- `y_pred` (*Tensor*) – The predicted output from the model.
- `y_true` (*Tensor*) – The ground truth or actual labels.

Returns
The calculated accuracy.

Return type
float

`calculate_precision(`*y_pred*, *y_true*`)`

Calculates the Recall of the model's prediction.

Parameters
- `y_pred` (*Tensor*) – The predicted output from the model.
- `y_true` (*Tensor*) – The ground truth or actual labels.

Returns
The calculated accuracy.

Return type
float

`calculate_recall(`*y_pred*, *y_true*`)`

Calculates the Recall of the model's prediction.

Parameters
- `y_pred` (*Tensor*) – The predicted output from the model.
- `y_true` (*Tensor*) – The ground truth or actual labels.

Returns
The calculated accuracy.

Return type
float

`call(`*inputs*, *training=False*`)`

The call function for the BaseModel.

Parameters
- `inputs` (*Tensor*) – The inputs to the model.
- `training` (*bool*) – A flag indicating whether the model is in training mode. Default is False.

Returns
None

Warning: This function must be implemented in the subclass, else it raises a NotImplementedError.

`eval_mode()`

Sets the model to evaluation mode.

`get_lr()`

Gets the current learning rate of the model.

Returns
The current learning rate.

Return type
float

`load_checkpoint(`*filepath*`)`

Loads the model weights from a checkpoint.

Parameters
`filepath` (*str*) – The file path where to load the model checkpoint from.

`optimize()`

Sets the model to training mode.

`save_checkpoint(`*filepath*`)`

Saves the model weights to a checkpoint.

Parameters
`filepath` (*str*) – The file path where to save the model checkpoint.

`step_scheduler()`

Adjusts the learning rate according to the learning rate scheduler.

`test_step(`*batch*`)`

Performs a test step using the input batch data.

Parameters
`batch` (*tuple*) – A tuple containing input data and labels.

Returns
The calculated loss, accuracy, and model predictions.

Return type
tuple

`train_mode()`

Sets the model to training mode.

`training_step(`*batch*`)`

Performs a training step using the input batch data.

Parameters
`batch` (*tuple*) – A tuple containing input data and labels.

Returns
The calculated loss and accuracy, labels and predictions

Return type
tuple

`validation_step(`*batch*`)`

Performs a validation step using the input batch data.

Parameters
`batch` (*tuple*) – A tuple containing input data and labels.

Returns
The calculated loss and accuracy.

Return type
tuple

`class models.tensorflow.models.CVTransferLearningModel(`**args: Any*, ***kwargs: Any*`)`

`__init__(`***kwargs*`)`

`call(`*inputs*, *training=True*`)`

The call function for the BaseModel.

Parameters
- `inputs` (*Tensor*) – The inputs to the model.

- `training` (*bool*) – A flag indicating whether the model is in training mode. Default is False.

Returns
None

> **Warning:** This function must be implemented in the subclass, else it raises a NotImplementedError.

`class models.tensorflow.models.HybridModel(`**args: Any*, ***kwargs: Any*`)`

`__init__(`***kwargs*`)`

`call(`*inputs*, *training=True*`)`

The call function for the BaseModel.

Parameters
- `inputs` (*Tensor*) – The inputs to the model.
- `training` (*bool*) – A flag indicating whether the model is in training mode. Default is False.

Returns
None

> **Warning:** This function must be implemented in the subclass, else it raises a NotImplementedError.

`class models.tensorflow.models.LSTMClassifier(`**args: Any*, ***kwargs: Any*`)`

LSTM-based Sequence Classifier

`__init__(`***kwargs*`)`

`call(`*inputs*`)`

Forward pass through the model

`class models.tensorflow.models.LSTMPredictor(`**args: Any*, ***kwargs: Any*`)`

`__init__(`***kwargs*`)`

`call(`*inputs*`)`

The call function for the BaseModel.

Parameters
- `inputs` (*Tensor*) – The inputs to the model.
- `training` (*bool*) – A flag indicating whether the model is in training mode. Default is False.

Returns
None

> **Warning:** This function must be implemented in the subclass, else it raises a NotImplementedError.

`class models.tensorflow.models.TransformerEncoderLayer(`**args: Any*, ***kwargs: Any*`)`

A Transformer Encoder layer as a subclass of tf.keras.layers.Layer.

Functionality: #. The class first initializes with key parameters for MultiHeadAttention and feedforward network. #. Then it defines the key components like multi-head attention, feedforward network, layer normalization, and dropout. #. In the call function, it takes input and performs self-attention, followed by layer normalization and feedforward operation.

Args:
d_model (int): The dimensionality of the input. n_head (int): The number of heads in the multi-head attention. dim_feedforward (int): The dimensionality of the feedforward network model. dropout (float): The dropout value.

Parameters
- `d_model` (*int*) – The dimensionality of the input.
- `n_head` (*int*) – The number of heads in the multi-head attention.
- `dim_feedforward` (*int*) – The dimensionality of the feedforward network model.
- `dropout` (*float*) – The dropout value.

Returns
None

Return type
None

Note: The implementation is based on the "Attention is All You Need" paper.

Warning: Ensure that the input dimension 'd_model' is divisible by the number of attention heads 'n_head'.

`__init__`(*d_model*, *n_head*, *dim_feedforward*, *dropout*)

`class models.tensorflow.models.TransformerPredictor`(**args: Any*, ***kwargs: Any*)

1 TransformerPredictor

A Transformer Predictor model that extends the BaseModel.

Functionality: #. The class first initializes with the learning rate and other parameters. #. It then creates an instance of TransformerSequenceClassifier. #. It also sets up the learning rate scheduler and the optimizer. #. In the call function, it simply runs the TransformerSequenceClassifier.

Args:
kwargs (dict): A dictionary of arguments.

param kwargs
A dictionary of arguments.

type kwargs
dict

returns
None

rtype
None

Note: The learning rate is set up with an exponential decay schedule.

Warning: The learning rate and gamma for the decay schedule must be specified in the 'kwargs'.

`__init__`(***kwargs*)

`call(`*inputs*, *training=True*`)`

The call function for the BaseModel.

Parameters

- `inputs` (*Tensor*) – The inputs to the model.
- `training` (*bool*) – A flag indicating whether the model is in training mode. Default is False.

Returns

None

Warning: This function must be implemented in the subclass, else it raises a NotImplementedError.

`class models.tensorflow.models.TransformerSequenceClassifier(`**args: Any*, ***kwargs: Any*`)`

A Transformer Sequence Classifier as a subclass of tf.keras.Model.

Functionality: #. The class first initializes with default or provided settings. #. Then it defines the key components like the transformer encoder layers and output layer. #. In the call function, it takes input and passes it through each transformer layer followed by normalization and dense layer for final output.

Args:

kwargs (dict): Any additional arguments. If not provided, defaults will be used.

Parameters

`kwargs` (*dict*) – Any additional arguments.

Returns

None

Return type

None

Note: The implementation is based on the “Attention is All You Need” paper.

Warning: The inputs should have a shape of (batch_size, seq_length, height, width), otherwise, a ValueError will be raised.

`__init__(`***kwargs*`)`

Chapter 14

Data Visualizations

`visualizations.visualize_augmentations(`*dataset*, *target_ sign*, *augmentation*, *n_ augments=4*`)`

Visualize *n_ augments* instances of a given target sign from the dataset and applies the augmentation provided.

This function generates a visual representation of the landmarks for each sample belonging to the specified *target_ sign* and applies the defined augmentation.

Args:
dataset (ASL_Dataset): The ASL dataset to load data from. target_sign (int): The target sign to visualize. augmentation: callable function to represent augmentation n_augments (int, optional): The number of samples to visualize. Defaults to 4.

Returns:
matplotlib.animation.FuncAnimation: A matplotlib animation object displaying the landmarks for each frame.

Parameters

- `dataset` (*ASL_Dataset*) – The ASL dataset to load data from.
- `target_sign` (*int*) – The target sign to visualize.
- `augmentation` – callable function with an augmentation
- `n_augments` (*int , optional*) – The number of samples to visualize, defaults to 6.

Type
callable

Returns
A matplotlib animation object displaying the landmarks for each frame.

Return type
matplotlib.animation.FuncAnimation

`visualizations.visualize_data_distribution(`*dataset*`)`

Visualize the distribution of data in terms of the number of samples and average sequence length per class.

This function generates two bar charts: one showing the number of samples per class, and the other showing the average sequence length per class.

Parameters
`dataset` (*ASL_Dataset*) – The ASL dataset to load data from.

`visualizations.visualize_target_sign(`*dataset*, *target_ sign*, *n_ samples=6*`)`

Visualize *n_ samples* instances of a given target sign from the dataset.

This function generates a visual representation of the landmarks for each sample belonging to the specified *target_ sign*.

Args:
dataset (ASL_Dataset): The ASL dataset to load data from. target_sign (int): The target sign to visualize. n_samples (int, optional): The number of samples to visualize. Defaults to 6.

Returns:

matplotlib.animation.FuncAnimation: A matplotlib animation object displaying the landmarks for each frame.

Parameters

- `dataset` (*ASL_Dataset*) – The ASL dataset to load data from.
- `target_sign` (*int*) – The target sign to visualize.
- `n_samples` (*int, optional*) – The number of samples to visualize, defaults to 6.

Returns

A matplotlib animation object displaying the landmarks for each frame.

Return type

matplotlib.animation.FuncAnimation

Chapter 15

Video Utilities

1 Video Utilities

This script provides a set of utility functions for video processing. It includes functions to convert Mediapipe results to a pandas DataFrame, capture frames from a video, draw landmarks on a frame, and convert frames to landmarks.

Imports: - Required libraries (pandas, numpy, cv2, mediapipe) - Constants from config file (FACE_FEATURES, POSE_FEATURES, HAND_FEATURES, ROWS_PER_FRAME)

`video_utils.capture_frames(`*video_path*, *target_sequence=None*`)`

> Capture frames from a video file.
>
> Args: video_path (str): The path to the video file. target_sequence (int): The target number of frames to capture. Defaults to None.
>
> Returns: frames (np.array): Array of captured frames.

`video_utils.convert_frames_to_landmarks(`*video_numpy_frames*`)`

> Convert frames to landmarks.
>
> Args: video_numpy_frames (np.array): Array of frames to convert.
>
> Returns: landmarks (np.array): Array of landmarks corresponding to each frame.

`video_utils.convert_mp_to_df(`*arr_results*`)`

> Convert MediaPipe results to a DataFrame.
>
> Args: arr_results (mp.Solutions): MediaPipe results object containing landmarks for face, pose, right and left hand.
>
> Returns: df_x (pd.DataFrame): DataFrame containing all landmarks.

`video_utils.draw_landmarks_on_frame(`*frame*, *results*`)`

> Draw landmarks on a frame.
>
> Args: frame (np.array): The frame to draw landmarks on. results (mp.Solutions): MediaPipe results object containing landmarks.
>
> Returns: frame (np.array): The frame with landmarks drawn.

Chapter 16

Metrics for evaluation

1 Generic Metric class

This module defines a generic Metric enumeration class. This class enumerates different metrics that can be used in evaluating the performance of a machine learning model. These metrics include Loss, Accuracy, F1Score, Recall, Precision, and AUC (Area under the curve of the ROC). It can be used in the context of machine learning model evaluation where one needs to switch between different metrics based on the problem at hand.

`class metrics.Metric(`*value*`)`

Enumeration class representing different metrics used in evaluation.

Members: - Loss: Represents the loss metric. - Accuracy: Represents the accuracy metric. - F1Score: Represents the F1 score metric. - Recall: Represents the recall metric. - Precision: Represents the precision metric. - AUC: Area under the curve of the ROC.

`AUC = 'AUC'`

`Accuracy = 'Accuracy'`

`F1Score = 'F1Score'`

`Loss = 'Loss'`

`Precision = 'Precision'`

`Recall = 'Recall'`

`__module__ = 'metrics'`

Chapter 17

Model Configurations

```yaml
models:
  TransformerPredictor:
    params:
      d_model: 192
      n_head: 8
      dim_feedforward: 512
      dropout: 0.001
      layer_norm_eps: !!float 1e-6
      norm_first: True
      batch_first: True
      num_layers: 3
      num_classes: 250
      learning_rate: 0.011
      gamma: 0.9
    data:
      augmentation_threshold: 0.05
      enableDropout: true
      augment: true
      load_additional_data: True
    optimizer:
      name: Adam
      params:
        lr: 0.001
        weight_decay: 0.001
    scheduler:
      name: ExponentialLR
      params:
        gamma: 0.92


  LSTMPredictor:
    params:
      input_dim: 192
      hidden_dim: 100
      layer_dim: 5
      output_dim: 250
      dropout: 0.5
    data:
      augmentation_threshold: 0.1
      enableDropout: true
      augment: true
```

(continues on next page)

(continued from previous page)

```
    optimizer:
      name: Adam
      params:
        lr: 0.001
        weight_decay: 0.0
    scheduler:
      name: ExponentialLR
      params:
        gamma: 0.9


  HybridModel:
    transformer_params:
      d_model: 192
      n_head: 8
      dim_feedforward: 512
      dropout: 0.001
      layer_norm_eps: !!float 1e-5
      norm_first: True
      batch_first: True
      num_layers: 4
    lstm_params:
      input_dim: 192
      hidden_dim: 512
      layer_dim: 4
      dropout: 0.001
    common_params:
      num_classes: 250
      learning_rate: 0.001
    data:
      augmentation_threshold: 0.1
      enableDropout: true
      augment: true
    optimizer:
      name: Adam
      params:
        lr: 0.001
        weight_decay: 0.0005
    scheduler:
      name: ExponentialLR
      params:
        gamma: 0.92


  TransformerEnsemble:
    TransformerSequenceClassifier:
      d_model: 192
      n_head: 8
      dim_feedforward: 2048
      dropout: 0.01
      layer_norm_eps: !!float 1e-5
      norm_first: true
      batch_first: true
      num_classes: 250
      learning_rate: 0.0011
    common_params:
```

(continues on next page)

(continued from previous page)

```
    num_classes: 250
    learning_rate: 0.011
    gamma: 0.93
    n_models: 5
  data:
    augmentation_threshold: 0.1
    enableDropout: true
    augment: true
  optimizer:
    name: Adam
    params:
      lr: 0.011
      weight_decay: 0.0
  scheduler:
    name: ExponentialLR
    params:
      gamma: 0.93


HybridEnsembleModel:
  TransformerSequenceClassifier:
    d_model: 192
    n_head: 8
    dim_feedforward: 2048
    dropout: 0.01
    layer_norm_eps: !!float 1e-5
    norm_first: True
    batch_first: True
    num_classes: 250
    learning_rate: 0.00106
  lstm_params:
    input_dim: 192
    hidden_dim: 512
    layer_dim: 4
    dropout: 0.05
  common_params:
    num_classes: 250
    learning_rate: 0.001
    n_models: 4
  data:
    augmentation_threshold: 0.1
    enableDropout: true
    augment: true
  optimizer:
    name: Adam
    params:
      lr: 0.001
      weight_decay: 0.001
  scheduler:
    name: ExponentialLR
    params:
      gamma: 0.95


CVTransferLearningModel:
  params:
```

(continues on next page)

(continued from previous page)

```
      num_classes: 250
    hparams:
      backbone: resnet152
      weights: null
      learning_rate: 0.001
      gamma: 0.95
    data:
      augmentation_threshold: 0.1
      enableDropout: true
      augment: true
    optimizer:
      name: Adam
      params:
        lr: 0.001
        weight_decay: 0.001
    scheduler:
      name: ExponentialLR
      params:
        gamma: 0.95



  YetAnotherTransformer:
    YetAnotherTransformerClassifier:
      learning_rate: 0.01
      d_model: 192
      embed_dim: 32
      n_head: 8
      expansion_factor: 4
      drop_rate: 0.0001
      attn_dropout: 0.0001
      num_layers: 2
      num_classes: 250
    common_params:
      num_classes: 250
      learning_rate: 0.001
      gamma: 0.9
    data:
      augmentation_threshold: 0.1
      enableDropout: true
      augment: true

  YetAnotherEnsemble:
    YetAnotherTransformerClassifier:
      learning_rate: 0.0011
      d_model: 192
      embed_dim: 64
      n_head: 8
      expansion_factor: 4
      drop_rate: 0.0001
      attn_dropout: 0.001
      num_classes: 250
    common_params:
      num_classes: 250
      learning_rate: 0.0011
      gamma: 0.95
```

(continues on next page)

(continued from previous page)

```
    n_models: 6
  data:
    augmentation_threshold: 0.01
    enableDropout: true
    augment: true
    load_additional_data: True
```

Python Module Index

Index