Human Action Recognition in the Dark: A Simple Exploration with Late Fusion and Image Enhancement

Anonymous authors
Paper under double-blind review

Abstract

Human action recognition (HAR) in low-light or dark environments presents unique challenges, often exacerbated by inadequate visibility and the absence of clear visual cues. While a biological eye can overcome such limitation by autotuning the sensitivity of retinal ganglion cells (RGC neurons) before feeding the visual data to the next layer down the pathway, immitating such a mechanism in an artifical neural network remains challenging under limited understanding of the human visual cortex. This paper aims to explore the possibilities of improving the performance of HAR with simple and explanable models through end-to-end training, with the hope of achieving the aforementioned close-to-human level of performance. Our study revolves around an innovative dataset and a modular pipeline.

1 Frame Sampling

To investigate the impact of different frame sampling strategies on Human Activity Recognition (HAR), we will consider videos from each category and compare the outcomes of uniform sampling and random sampling.

Like in our previous study, each of the frame samplers is implemented as an information processing unit (`keras.Model` or `keras.layers.Layer`). Each unit takes a batch of frame sequences of the video within the dataset and outputs a batch of the sampled frames.

The pros and cons of each sampling techniques are as follows:

- Uniform Sampling
  - Pros

    Equal Representation It ensures that each frame in the video has an equal chance of being included in the training set, which prevents bias towards certain actions over time periods.

    Consistent Data Distribution It helps maintain a consistent distribution of samples across different classes and time periods, which can be important for achieving a balanced and representative training set.
  - Cons

    Irrelevant Information It may lead to the inclusion of redundant or less informative frames while potentially missing crucial frames that capture important actions. This could be a limitation if certain actions are rare but critical.

    Ignored Temporal Dynamics In some cases, actions may have varying temporal dynamics, and uniform sampling might not capture the full range of motion or transitions between frames.
- Random Sampling
  - Pros

    Temporal Variability It's capable of capturing the temporal variability of actions, ensuring that the model is exposed to different instances of actions

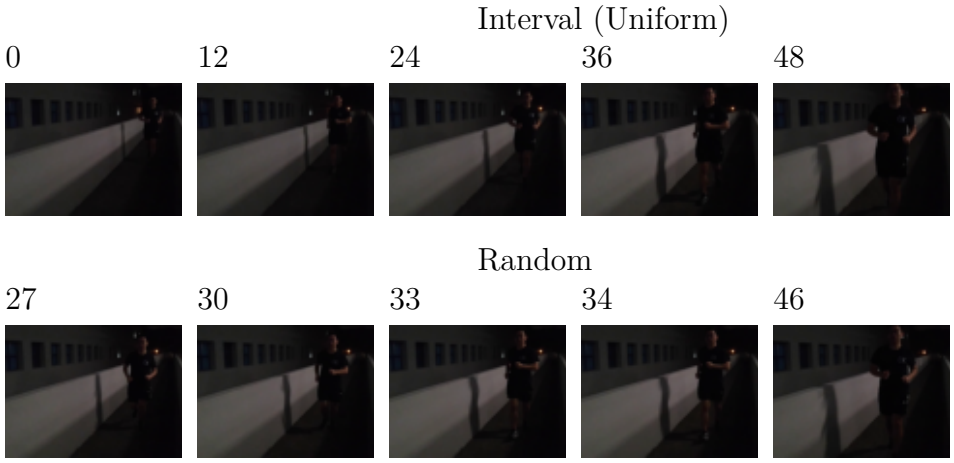


Figure 1: Sampled frames of a video under the '`Run`' category (encoded as `1`). As can be seen, the acitivity has recurring patterns.

across time. This can be important for recognizing actions with varying speeds or durations.

Interoperability with Models Combined with the shuffling of batches, it makes the input to the model more diverse. This prevents overfitting.

– Cons

Reproducibility It makes experiments less reproducible, as each run may yield a different set of training samples. This can make it challenging to compare results across different runs.

As illustrated in Figure 1, the activities involved in our dataset, including jump, run, sit, stand, turn, and walk, are inherently mundane and exhibit no significant temporal variations within the spatial domain. In scenarios where the actions are relatively straightforward and lack pronounced temporal dynamics, uniform sampling is a pragmatic choice for the initial sampling strategy. The author theorizes that uniform sampling would not hurt the downstream model's generalizability.

2 Feature Extraction

The feature extraction module is designed to extract valuable spatial information from each frame (or image) in the video dataset while preserving the temporal relationships between frames.

2.1 Model: ConvNeXt

A suitable model for such task is a time distributed 2D Convolutional Neural Network (CNN), bearing a 3D structure.

ConvNeXt, a notable example of the latest CNN-based models, "compete(s) favorably with Transformers in terms of accuracy and scalability, achieving 87.8% ImageNet top-1 accuracy" (Liu et al., 2022). ConvNeXt retains the simplicity and efficiency of standard CNNs while still manages to be accurate. As its trained and fine-tuned on a large dataset (ImageNet), it is particularly suitable for a video classification task, where the features are oftentimes diverse. The version of ConvNeXt selected for this experiment, ConvNeXt-S (small), has 50 million parameters and an impressive 85.8% top-1 accuracy (Liu et al., 2022).

As for the implementation, `tensorflow.keras`'s implementation of ConvNeXt incorporates builtin data preprocessing and output postprocessing, which eliminates the need for dedi-

cated preprocessing and postprocessing layers. The general pipeline of data processing in ConvNeXt is as follows:

Normalization This normalizes the pixel data of each frame on the last axis, which is the channel axis.

Transformation This transforms the normalized input data using the network's pretrained weights.

Pooling By the `pooling='avg'` option, the network performs postprocessing by global average pooling, which effectively reduces the output dimension.

Like every information module in this paper, ConvNeXt supports batch processing.

2.2 Experiment: Frame Sampling + Feature Extraction

The sampled frame feature extraction process is characterized by the following input and output specifications:

Input Shape:

- Batch Size
- Frame Count
- Frame Width
- Frame Height
- Frame Channel Count

Output Shape:

- Batch Size
- Frame Count (result of frame sampling)
- Feature Count (result of feature extraction)

It is important to note that the batch size remains constant throughout the operation. During the process, each frame undergoes conversion into a feature vector, the shape of which is determined by the feature extraction module. Additionally, it is expected that the frame count will be reduced as a result of frame sampling.

To illustrate –

For a ragged input of shape `[2, '<variable>', '<variable>', '<variable>', 3]`, the shape of the output is `[2, 8, 768]`, where `<variable>` represents a variable length.

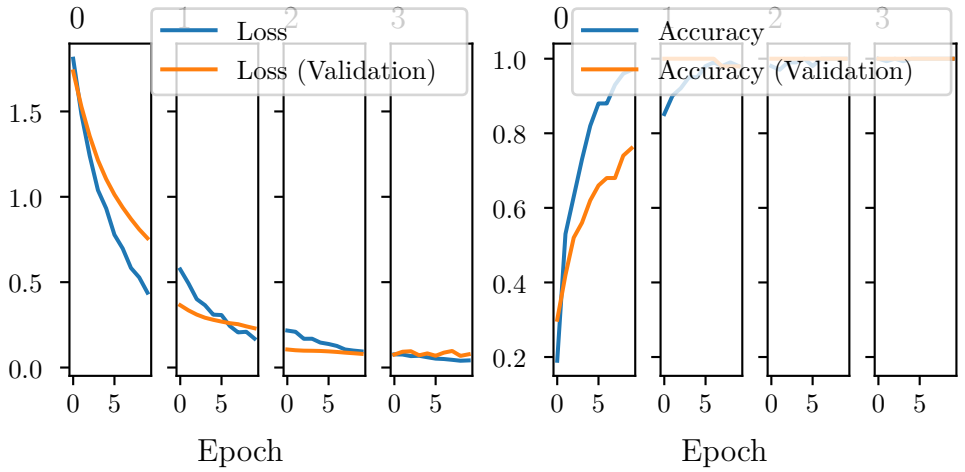
2.3 Discussion: Improvements

Apparently, our features could be further localized. From an extracted bounding box of interest, a much more explainable model exists. Psychological studies have found that humans are able to perform action recognition by merely looking at a set of moving points on a body (Johansson, 1973), indicating that key aspects of body language could very well be encoded in these keypoints. One example of such a system is MoveNet, which accurately identifies and tracks these keypoints. In our CNN-only model, we use these keypoints (pixels) more directly. However, for MoveNet, instead of combining them at a later stage (late fusion), we input them into an embedding layer at an earlier stage. This method allows for a more nuanced processing of "body language", enabling our model to interpret and analyze human movements with greater accuracy and detail.

3 Classifier Training and Evaluation

3.1 Architecture

The classifier goes right after the feature extraction module in the classification pipeline.



(a) The training losses over time.

(b) The training accuracies over time.

Figure 2: The curve of diminishing loss and escalating accuracy on the *training* set. Despite some minor fluctuations, signs of overfitting are nowhere to be found; yet, our model still performs poorly on the *validation* set.

Given that our video data exhibits long-term dependencies in a naturally sequential manner, the logical choice for addressing this type of problem is Long Short-Term Memory (LSTM). In our implementation, we utilize the LSTM module (`keras.layers.LSTM`), followed by a dense layer (`keras.layers.Dense`) for the purpose of multiclass classification. The downstream process involves performing softmax through the loss evaluation method, specifically using `keras.losses.SparseCategoricalCrossentropy` with `from_logits=` ↩ ↪ `True`. The dense layers between LSTM and the final layer are omitted, as a simplified, consolidated architecture keeps the problem of catastrophic overfitting at bay, particularly in the case where the training dataset is small.

The pros and cons of LSTMs include:

- Pros

  Long-Term Dependency Handling LSTM effectively puts sequential data in context over extended periods of time. This is crucial for our experiment as human actions are better interpreted in broad, time-domain based contexts.

  Information Retention Much like the hippocampus in a brain, LSTMs utilize memory cells, which allow them to selectively remember or forget information. This ability is useful for tasks where retaining *relevant* information over time is important. Over-rembering irrelevant information can lead to overfitting.

- Cons

  Computational Complexity Since LSTM is a deep recurrent neural network, it's computationally expensive.

  Interpretability LSTMs, like many deep learning models, are often considered as "black-box" models. Understanding the internal representations and decision-making processes of LSTMs can be challenging, which may be a drawback in certain applications where interpretability is crucial.

  Overfitting LSTMs, like almost all deep learning models, are prone to overfitting, especially when dealing with small datasets. Regularization techniques and careful tuning of hyperparameters are possible, but they will only fix the problem to a certain extent.

3.2 Training

For training, cross-validation is done on the above base model.

3.2.1 Fitting

The model is trained under the following parameters:

Epochs Due to the use of cross-validation, we use a small number of epochs (iterations) for training within each cross-validation session.

Batch Size The batch size is proportional to the number of classes in the dataset. This is done to ensure that the model receives a representative sample from each class during training, promoting better generalization to diverse class distributions.

Optimizer The optimizer selected for our model is Adaptive Moment Estimation (Adam) for its adaptive learning rate. It prevents the optimizer from getting stuck in local minima and accelerates convergence.

Early Stopping The loss of our model is monitored throughout the training period; once it does not increase over a certain number of epochs, it stops training to prevent overfitting.

3.2.2 Cross-validation

As the amount of data for training is extremely limited, stratified k-Fold cross validation is used to ensure a better coverage of the data. Upon initial training and evaluation, the author has observed a severe problem of overfitting; although the issue was somehow mitigated by adding dropout layers within the LSTM and in-between the subsequent layers, the testing accuracy was still not high. Therefore, the aforementioned cross validation technique was introduced.

The general procedure of cross-validation is as follows:

Data Splitting The entire dataset is randomly divided into k equal-sized folds.

Model Training and Validation This process is repeated k times, with each of the k folds serving as the validation set once, and the remaining $k-1$ folds forming the training set.

- In the first iteration, the first fold is the validation set, and the remaining folds are the training set.
- This process is repeated until each fold has served as the validation set once.

Performance Measurement After each training and validation on a fold, the model's performance is measured using metrics like accuracy or precision. This gives a performance score for each fold. The metrics are subsequently used to update the model, if applicable. In our experiment, the validation loss is used to update the weights of our model through backpropagation.

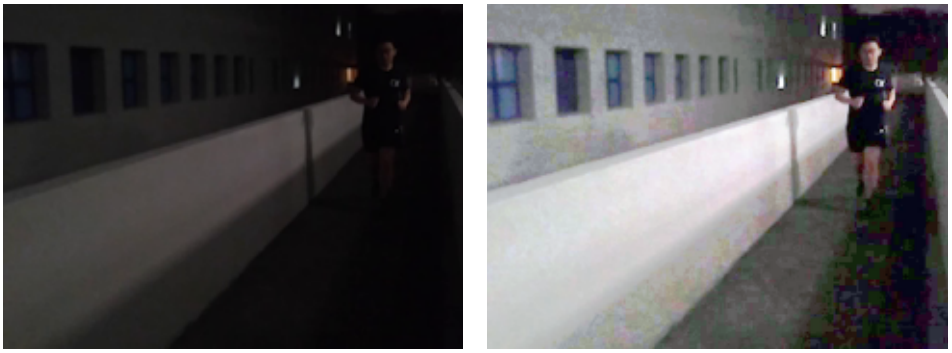
Implementation wise, our cross-validator makes use of the `tensorflow_dataset` API and generates data iteratively; the stratification of our data is automatically implemented by `tensorflow_dataset`'s batch sampling mechanism, where the proportions of each class within the dataset are preserved with the best effort possible for each batch.

3.3 Evaluation

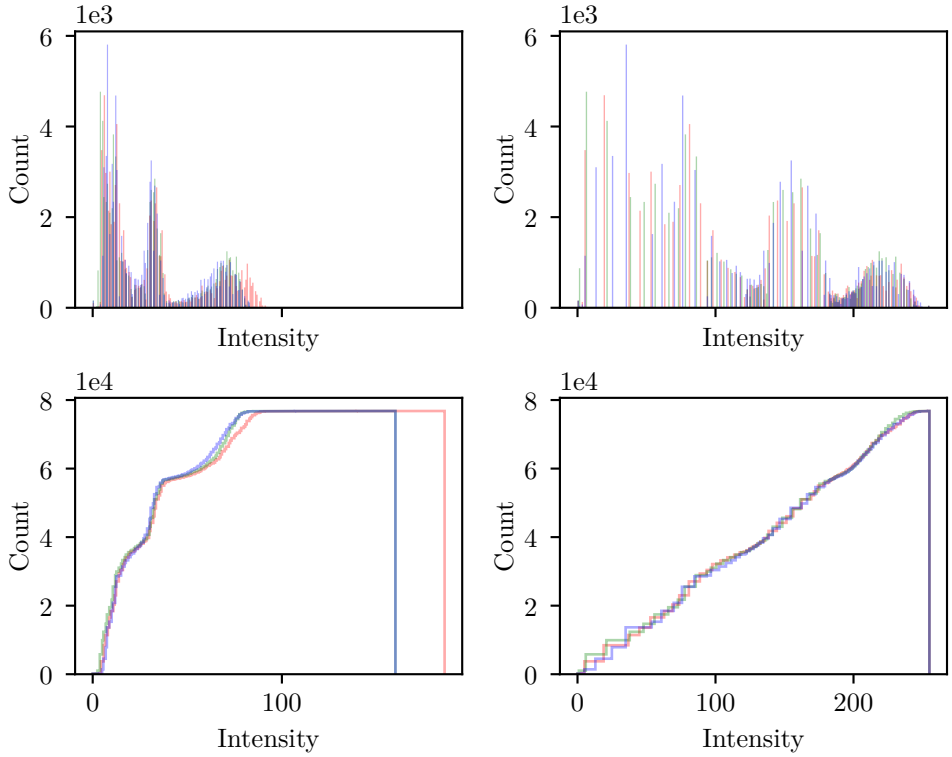
The performance of the model is evaluated on the validation dataset by categorical accuracy.

The model has a loss of 2.424560546875 and an accuracy of 0.1145833358168602.

Apparently, this is not ideal. Although our feature extraction module is pretrained, its weights are all likely derived from the features in non-challenging environments, allegedly the ones with good illumination. We theorize that the classifier's performance will increase once a means of illuminance enhancement is applied.



(a) The image before (left) and after (right) the equalization. Notice the dramatic improvement in brightness and contrast, as well as the reduction in overall visual quality. This can be attributed to the dramatic changes in the image's color balance, the over-amplification of artifacts present in the original image.



(b) The histograms (top) and CDFs (bottom) of the images before (left) and after (right) the equalization. The distributions of pixel intensities become roughly normal as expected.

Figure 3: The effect of histogram equalization.

4 Classifier Training and Evaluation with Image Enhancements

4.1 Histogram Equalization

In dark conditions, images often suffer from poor contrast, making it difficult to distinguish between the subject and the background. Histogram equalization comes to rescue by adjusting the brightness levels across the image, enhancing the contrast and making the human figures more distinguishable from their surroundings.

Our deep learning model relies heavily on the extraction of features such as edges, shapes, and textures. By improving the contrast of the images, we theorize that histogram equalization could make these features more pronounced and easier to detect, leading to more accurate localization of human figures and recognition of human actions.

Although the original algorithm only works on single-channel images, it can be extended by running equalization on the channels individually and piece together the end results to form an enhanced multi-channel image. In `tensorflow`, this is already available as an equalization layer, `keras_cv.layers.Equalization`.

The algorithm works as follows:

Histogram Normalization For a grayscale image, the histogram $H(i)$ represents the frequency of each intensity level. It's defined as

$$H(i) = \text{Number of pixels with intensity } i$$

Where

- i the intensity level.

The histogram is then normalized, through any statistically plausible normalization technique such as mean normalization:

$$h(i) = \frac{H(i)}{N}$$

Where

- N the total number of pixels.

CDF (Cumulative Distribution Function) Calculation The CDF, $C(i)$, accumulates the sum of the histogram values up to a certain intensity level. It's defined as:

$$C(i) = \sum_{j=0}^{i} h(j)$$

Where

- i the intensity level.

Equalization The histogram equalization maps the original intensity levels to new levels. The new intensity level $I_{\text{new}}(i)$ is calculated by:

$$I_{\text{new}}(i) = \frac{(L-1) \cdot C(i)}{N}$$

Where

- L the number of possible intensity levels.

Each pixel in the original image with intensity i is then mapped to a new intensity $I_{\text{new}}(i)$. This results in an image with a more uniform distribution of intensities (Figure 3).

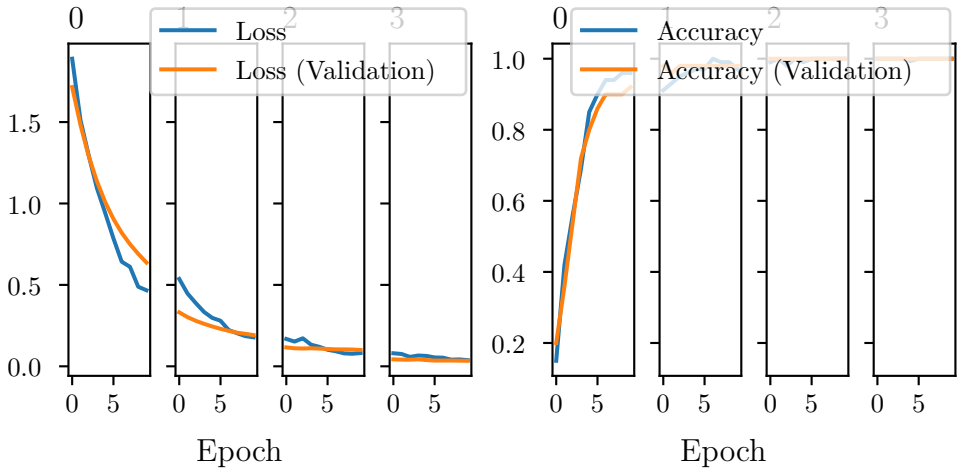
4.2 Training

The training of the model with image enhancement follows the same procedure described in the previous section.

4.3 Evaluation

Through image enhancement, the model has achieved a loss of 2.2525634765625 and an accuracy of 0.125.

Our experiment results shows that histogram equalization does enhance the performance of the CNN-based frame sequence classification model for this *specific* dataset, albeit to an insignificant extent. It's crucial to recognize that this doesn't necessarily imply a universal performance boost with histogram equalization. Since the amount of data for training is *extremely* limited, more data is needed for a more comprehensive evaluation of the experiment.



(a) The training losses over time.

(b) The training accuracies over time.

Figure 4: The curve of diminishing loss and escalating accuracy, again. Image enhancement is used with the hope of improving the model.

5 End-to-end Training

The entire project is implemented with simplicity and modularity in mind, under the influence of the UNIX principle. Therefore, all modules (layers and models) within our project is automatically end-to-end, with end-to-end training enabled by chaining the modules, i.e. the information processing units, together through `keras.layers.Sequential`.

In terms of performance, our end-to-end model exhibits higher time complexity compared to its distributed non-end-to-end counterparts, primarily because the features must be regenerated with each iteration. However, this challenge is not insurmountable. By employing caching, distributed computing, and parallelization strategies, we can harness the computational prowess of multiple machines or processing units to alleviate the complexities posed by increased computational demands.

Conversely, this approach eliminates the necessity to explicitly store features, thereby reducing the runtime storage complexity of the model. Through thoughtful implementation of these strategies, we can strike a balance between computational efficiency and storage requirements, optimizing the overall performance of our model.

For details on the training and evaluation procedure, refer to the code in Appendix A.

References

Gunnar Johansson. Visual perception of biological motion and a model for its analysis. *Perception & Psychophysics*, 14(2):201–211, Jun 1973. ISSN 1532-5962. doi: 10.3758/BF03212378. URL `https://doi.org/10.3758/BF03212378`.

Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s, 2022.

A Appendix

The code used to run the experiment, written in a Jupyter notebook. An IPython interpreter is required. Additional code for parsing the dataset is available upon request.

A.1 Setup

The full project is available at: - https://github.com/379f523e295b/ee6222-assignment1-fc9a

(NOTE: The full project structure is required to run this notebook)

A.1.1 Packages

Uncomment below to install. Utitilties

```
#%pip install --quiet jupyter-spaces
#%pip install --quiet scrapbook scrapbook-ext==0.0.0a2
```

Frameworks

```
#%pip install --quiet matplotlib
#%pip install --quiet keras-cv
#%pip install --quiet tensorflow tensorflow-datasets
#%pip install --quiet pydot
```

Dataset (EE6222 HAR Video)

... requires `./datasets` present

```
%pip install --quiet --editable ./datasets
```

A.1.2 Extensions

... for Jupyter Notebooks

```
%load_ext jupyter_spaces
```

A.1.3 Imports and Utilities

`IPython`: ... for HTML rendering

```
import IPython as ipy
```

`scrapbook`: ... for data storage inside notebooks

```
import scrapbook
import scrapbook_ext.encoders.pickle

scrapbook_ext.encoders.pickle.load()
```

`matplotlib`: ... for data visualization

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rcParams.update({
    # text
    #'text.usetex': True,
    'font.family': 'sans-serif',
    # figure
    'figure.constrained_layout.use': True,
```

```
    # pgf
    'pgf.texsystem': 'lualatex',
    'pgf.rcfonts': False,
    'pgf.preamble': r'\renewcommand\sffamily{}'
})
```

`tensorflow`: ... for data processing and learning

```
import tensorflow as tf
import tensorflow_datasets as tfds

from tensorflow import keras

# enable mixed precision for performance
keras.mixed_precision.set_global_policy('mixed_float16')
```

`tensorflow_datasets_ee6222har`: ... `tensorflow` dataset

```
import tensorflow_datasets_ee6222har.ee6222har as tfds_ee6222har

ds_builder = tfds_ee6222har.builder.Builder()
ds_builder.download_and_prepare()
```

```
ds = ds_builder.as_dataset(as_supervised=False)
```

Miscellaneous

```
take_one = lambda x: next(iter(x))
```

A.2 EDA

```
%%space demo:eda

import numpy as np

targets, freqs_target = np.unique(
    np.asarray(list(ds['train'].map(lambda x: x['label']))),
    return_counts=True,
)

fig, ax = plt.subplots()
ax.barh(
    [
        ds_builder.info.features['label'].mapping[target]
        for target in targets
    ],
    freqs_target
)
fig.show()
```

A.3 Frame Sampling

```
import abc
import typing


class BaseSampling(keras.layers.Layer, abc.ABC):
```

```
    # TODO NOTE index sampling
    # TODO NOTE range [minval, maxval)
    def _range_sampler(
        self,
        range_n: typing.Tuple[int, int],
        n_samples: int,
    ) -> typing.Sequence[tf.int32]:
        raise NotImplementedError

    # TODO NOTE index sampling
    # TODO NOTE range [minval, maxval)
    def _sample_range(
        self,
        range_n: typing.Tuple[int, int],
        n_samples: int | float,
    ) -> typing.Sequence[tf.int32]:
        # TODO
        minval, maxval = range_n
        l_range = maxval - minval + 1
        return self._range_sampler(
            range_n=range_n,
            n_samples=(
                tf.cast(
                    self.n_samples * tf.cast(l_range, tf.float32),
                    tf.int32,
                )
                if isinstance(n_samples, float) else
                (n_samples)
            ),
        )

    def _sample_indices(self, input):
        axis = self.axis
        n = tf.shape(input)[axis]
        return self._sample_range(
            range_n=(0, n),
            n_samples=self.n_samples,
        )

    def _gather_samples(self, input, indices):
        return tf.gather(
            input,
            indices=indices,
            axis=self.axis,
        )

    def __init__(
        self,
        n_samples: float | int,
        axis: int,
        **kwargs,
    ):
        super().__init__(**kwargs)
        self.axis = axis
        self.n_samples = n_samples

    def call(self, input):
        return self._gather_samples(
            input,
            indices=self._sample_indices(
                input
            ),
        )

class RandomSampling(BaseSampling):
```

```
    def __init__(
        self,
        *args,
        seed: int = None,
        **kwargs,
    ):
        super().__init__(*args, **kwargs)
        self.seed = seed

    # TODO NOTE returns indices of xs
    def _uniform_choice(
        self,
        xs, n_samples,
        unique=True,
        **kwargs
    ):
        return (
            tf.random.uniform_candidate_sampler(
                [range(0, len(xs))],
                len(xs),
                num_sampled=n_samples,
                unique=unique,
                range_max=len(xs),
                seed=self.seed,
                **kwargs,
            ).sampled_candidates
        )

    def _range_sampler(
        self,
        range_n: typing.Tuple[int, int],
        n_samples: int,
    ):
        minval, maxval = range_n
        xs = tf.range(
            minval, maxval,
            dtype=tf.int32
        )
        return tf.gather(
            xs,
            tf.sort(
                self._uniform_choice(
                    xs,
                    n_samples=n_samples,
                    unique=True,
                ),
                direction='ASCENDING',
            ),
        )


class IntervalSampling(BaseSampling):
    # NOTE linspace with stop excluded
    # NOTE align the min/max for fixed int intervals
    def _linspace_openend(
        self,
        start, stop,
        num,
        base=1
    ):
        _linspace_impl = (
            lambda start, stop, num, base:
            tf.linspace(
                start + start % base,
                stop - stop % base,
```

```
                num=num,
            )
        )
        return _linspace_impl(
            start,
            stop - base,
            num=num,
            base=base,
        )

    def _range_sampler(
        self,
        range_n: typing.Tuple[int, int],
        n_samples: int,
    ):
        minval, maxval = range_n
        return tf.cast(
            self._linspace_openend(
                minval, maxval,
                num=n_samples,
                # NOTE align the min/max with the number of intervals
                base=(n_samples - 1),
            ),
            dtype=tf.int32,
        )
```

A.3.1 Model: Interval Sampling

```
class BatchIntervalSampling(keras.layers.Layer):
    def __init__(self, num_samples, input_axis=0, **kwargs):
        super().__init__(**kwargs)
        self.num_samples = num_samples
        self.input_axis = input_axis

    def call(self, inputs):
        # ref https://stackoverflow.com/a/64436208
        @tf.function
        def _impl_array_slice(
            a, axis,
            start, stop,
            step=1,
        ):
            return a[
                (slice(None),) * (axis % a.ndim)
                + (slice(start, stop, step),)
            ]

        @tf.function
        def _impl_array_sample(
            a, axis,
            n_samples,
            start, stop,
        ):
            # NOTE first + n * step == last
            _aligned_interval = lambda first, last, n: (
                first,
                last - (last - first) % n,
                (last - first) // n,
            )

            first, last, step = _aligned_interval(
                start, stop - 1,
                # NOTE number of intervals
```

```
                n=n_samples - 1
            )

            return _impl_array_slice(
                a, axis=axis,
                start=first, stop=last + 1,
                step=step,
            )

        @tf.function
        def _array_sample(
            a, axis,
            n_samples
        ):
            return _impl_array_sample(
                a, axis,
                n_samples=n_samples,
                start=0, stop=tf.shape(a)[axis],
            )

        @tf.function
        def _unrag(input):
            if not isinstance(input, tf.RaggedTensor):
                return input
            return input.to_tensor()

        output_shape = inputs.shape.as_list()
        output_shape[self.input_axis] = self.num_samples

        return tf.map_fn(
            lambda input: _array_sample(
                _unrag(input), axis=self.input_axis,
                n_samples=self.num_samples,
            ),
            inputs,
            fn_output_signature=tf.TensorSpec(
                shape=output_shape[1:],
                dtype=inputs.dtype
            ),
        )
```

A.3.2 Model: Random Sampling

```
# NOTE see above
```

A.3.3 Demo: Uniform, Random

```
%%space demo:frame_sampling

import typing

n_samples = 5
d = take_one(ds['train'])

fig = plt.figure()
subfigs = fig.subfigures(nrows=2, ncols=1).flat

for fig_, suptitle_, sampler_cls_ in [
    (subfigs[0], 'Interval (Uniform)', IntervalSampling),
    (subfigs[1], 'Random', RandomSampling),
]:
```

```
    sampler_cls_: typing.Type[BaseSampling]

    fig_.suptitle(suptitle_, horizontalalignment='left')
    frame_indices = sampler_cls_(
        n_samples=n_samples,
        axis=0,
    )._sample_indices(d['frameseq'])

    axes = fig_.subplots(
        nrows=1,
        ncols=len(frame_indices)
    )
    for ax, frame_idx in zip(
        axes.flat,
        frame_indices,
    ):
        ax.axis('off')
        ax.imshow(d['frameseq'][frame_idx])
        ax.set_title(int(frame_idx), loc='left')

fig.show()
scrapbook.glue(
    'figure:frame_sampling',
    fig,
    encoder='pickle'
)
scrapbook.glue(
    'data:frame_sampling',
    {
        'label': d['label'].numpy(),
        'label:name':
            ds_builder.info.features['label']
            .decode_example(
                d['label']
            )
    }
)
```

A.4 Feature Extraction

... through late fusion

```
class BatchStack(keras.layers.Layer):
    def __init__(
        self,
        layer: keras.layers.Layer,
        **kwargs
    ):
        super().__init__(**kwargs)
        self.layer = layer

    def call(self, inputs):
        return tf.map_fn(self.layer, inputs)

class Unragging(keras.layers.Layer):
    def call(self, inputs):
        if not isinstance(inputs, tf.RaggedTensor):
            return inputs
        return inputs.to_tensor()
```

```
class Flattened(keras.layers.Layer):
    def __init__(self, layer, axis, **kwargs):
```

```
        super().__init__(**kwargs),
        self.layer = layer
        self.axis = axis

    def call(self, inputs):
        axis = self.axis
        outputs = tf.reshape(
            inputs,
            tf.concat([
                [-1],
                tf.shape(inputs)[(axis + 1):],
            ], axis=0)
        )
        outputs = self.layer(outputs)
        outputs = tf.reshape(
            outputs,
            tf.concat([
                tf.shape(inputs)[:(axis + 1)],
                tf.shape(outputs)[1:],
            ], axis=0)
        )
        return outputs
```

A.4.1 Model: ConvNeXt

```
def ConvNeXtFeatureExtraction(
    trainable=False,
    name='model.feature.convnext',
    **convnext_kwds
):
    model = keras.Sequential([
        Unragging(),
        keras.layers.TimeDistributed(
            # NOTE For ConvNeXt, preprocessing is included
            # in the model using a Normalization layer.
            # ConvNeXt models expect their inputs to be
            # float or uint8 tensors of pixels with values in the ↩
                ↪ [0-255] range.
            # NOTE ref https://keras.io/api/applications/convnext/# ↩
                ↪ convnextsmall-function
            base_model := keras.applications.ConvNeXtSmall(
                include_top=False,
                weights='imagenet',
                pooling='avg',
                **convnext_kwds,
            ),
            name=name,
        )
    ], name=name)
    base_model.trainable = trainable
    return model
```

A.4.2 Model: MoveNet

```
import functools

class MoveNet(keras.Model):
    class _repository:
        import tensorflow_hub as tfhub
        base_url = 'https://tfhub.dev/google/movenet/singlepose'
        variant_url = {
```

```
            'lightning': rf'{base_url}/lightning/4',
            'thunder': rf'{base_url}/thunder/4',
        }

        @classmethod
        @functools.cache
        def load(cls, variant=None):
            return cls.tfhub.load(
                cls.variant_url.get(variant, 'thunder')
            )

    @classmethod
    def from_hub(cls, variant=None):
        return cls(
            cls._repository.load(variant)
        )

    # NOTE saved_model: tensorflow saved model
    def __init__(self, saved_model, **kwargs):
        super().__init__(**kwargs)
        self.model = saved_model

    @property
    def base(self):
        return self.model.signatures['serving_default']

    # NOTE input shape (1, w, h, c) or (w, h, c)
    def call(self, inputs):
        _n_batches, *input_frame_size, _n_channels = (
            self.base.inputs[0].shape
        )
        inputs = tf.image.resize_with_pad(
            inputs,
            *input_frame_size
        )
        # NOTE movenet's batch processing is dysfunctional
        inputs = tf.reshape(inputs, self.base.inputs[0].shape)
        inputs = tf.cast(inputs, dtype=self.base.inputs[0].dtype)
        return self.base(inputs)['output_0']
```

```
def MoveNetFeatureExtraction(
    model: MoveNet | None = None,
    name='model.feature.movenet',
):
    if model is None:
        model = MoveNet.from_hub()

    return keras.Sequential([
        Unragging(),
        # ensure input is tf.float32
        keras.layers.Lambda(lambda x: tf.cast(x, tf.float32)),
        keras.layers.TimeDistributed(
            BatchStack(
                keras.Sequential([
                    model,
                    # NOTE excl. last axis (probability/confidence)
                    keras.layers.Lambda(lambda x: x[..., :-1]),
                    keras.layers.Lambda(tf.squeeze),
                ])
            )
        ),
        # TODO
        keras.layers.TimeDistributed(
            keras.layers.AveragePooling2D()
```

```
        ),
        # TODO embedding layers
    ], name=name)
```

Demo: Frame Sampling + Feature Extraction

```
%%space demo:feat_extraction

def DemoFeatureExtraction(n_frames=8):
    return (
        keras.models.Sequential([
            BatchIntervalSampling(num_samples=n_frames),
            Unragging(),
            ConvNeXtFeatureExtraction(),
        ])
    )

batch_size = 2
sample_input = take_one(ds['train'].ragged_batch(batch_size))['frameseq']

def _descr_shape(
    shape: tf.TensorShape,
    placeholder: str
):
    return [
        n if n is not None else placeholder
        for n in shape
    ]

placeholder = '<variable>'
report = {
    'placeholder:shape': placeholder,
    'shape:input': _descr_shape(
        sample_input.shape,
        placeholder=placeholder,
    ),
    'shape:output': _descr_shape(
        DemoFeatureExtraction()(sample_input).shape,
        placeholder=placeholder,
    ),
}

scrapbook.glue(
    'data:feat_extraction',
    report
)
report
```

A.5 Classification

```
class BaseSplitter:
    def __call__(
        self,
        builder: tfds.core.DatasetBuilder
    ) -> typing.Iterable[typing.Tuple]:
        raise NotImplementedError

class KFoldSplitter:
    def __init__(
        self,
```

```
        subset: str,
        n_splits: int
    ):
        self.subset = subset
        self.range_perct = range(0, 100, 100 // n_splits)

    def __call__(self, builder: tfds.core.DatasetBuilder):
        # ref https://www.tensorflow.org/datasets/splits#↩
            ↪cross_validation
        next_k = lambda k: min(
            k + self.range_perct.step,
            self.range_perct.stop
        )
        splits_fold = [
            [f'{self.subset}[:{k}%]'
             f'+{self.subset}[{next_k(k)}%:]'
                for k in self.range_perct],
            [f'{self.subset}[{k}%:{next_k(k)}%]'
                for k in self.range_perct],
        ]
        datasets = [
            builder.as_dataset(split=split_fold)
            for split_fold in splits_fold
        ]
        return zip(*datasets)

def cross_validate(
    model: keras.Model,
    builder: tfds.core.DatasetBuilder,
    splitter: BaseSplitter,
    batch_fn: typing.Callable[
        [tf.data.Dataset],
        tf.data.Dataset
    ] = None,
    **fit_kwds,
):
    def _impl(
        model: keras.Model,
        splitter: typing.Iterable[typing.Tuple],
        **fit_kwds,
    ):
        return [
            model.fit(
                ds_train,
                validation_data=ds_val,
                **fit_kwds,
            ) for ds_train, ds_val in splitter
        ]

    return _impl(
        model=model,
        splitter=(
            [
                batch_fn(ds) if batch_fn is not None else ds
                for ds in dss
            ] for dss in splitter(builder)
        ),
        **fit_kwds,
    )
```

A.5.1 Classifier

```
def LSTMClassifier(n_classes, name=None, **seq_kwds):
```

```
    return keras.models.Sequential(
        [
            keras.layers.LSTM(
                128,
                recurrent_dropout=.1,
            ),
            keras.layers.Dropout(.25),
            keras.layers.Dense(n_classes),
        ],
        name=(
            name if name is not None else
            rf'model.classifier_{n_classes}_class'
        ),
        **seq_kwds
    )
```

A.5.2 Model

`Model`: The model factory method to enable end-to-end training.

```
def Model(
    n_classes,
    augmentations: typing.Sequence[keras.layers.Layer] = [],
    feature_extraction: typing.Callable[[], keras.layers.Layer] | None ↩
        ↪ = None,
    name: str = 'model',
    compile: bool = True,
):
    if feature_extraction is None:
        feature_extraction = ConvNeXtFeatureExtraction()
    res = keras.models.Sequential([
        # NOTE frames are expected to be multi-channel 8-bit
        keras.layers.Lambda(lambda x: tf.cast(x, tf.uint8)),
        BatchIntervalSampling(num_samples=32),
        Unragging(),
        *augmentations,
        feature_extraction,
        LSTMClassifier(n_classes=n_classes),
    ], name=name)

    if compile:
        res.compile(
            optimizer=keras.optimizers.Adam(
                learning_rate=1e-4,
            ),
            loss=keras.losses.SparseCategoricalCrossentropy(
                from_logits=True,
            ),
            metrics=['accuracy'],
        )

    return res
```

`ModelManager`: A helper class that manages a `Model` instance to enable training and evaluation in one take.

```
class ModelManager:
    class TrainingResult:
        import matplotlib as mpl
        import matplotlib.pyplot as plt

        def __init__(
```

```
            self,
            histories: typing.Sequence[keras.callbacks.History]
        ):
            self.histories = histories

        def plot(
            self,
            fig: mpl.figure.Figure | None = None,
            metrics: typing.Collection[str] | None = None,
            metric_labels: typing.Mapping[str, str] | None = None,
            **subplot_kwds,
        ):
            metrics = metrics or ['loss', 'val_loss']
            if not metric_labels:
                metric_labels_base = {
                    'loss': 'Loss',
                    'accuracy': 'Accuracy',
                }
                metric_labels = {
                    **metric_labels_base,
                    **{f'val_{key}': f'{value} (Validation)'
                        for key, value in metric_labels_base.items()}
                }

            fig = fig or self.plt.gcf()
            axs = fig.subplots(
                nrows=1, ncols=len(self.histories),
                sharex=True, sharey=True,
                **subplot_kwds,
            )

            for n, (ax, history) in enumerate(
                zip(axs.flat, self.histories)
            ):
                for metric in metrics:
                    if metric not in history.history:
                        continue
                    ax.plot(
                        history.epoch,
                        history.history[metric],
                        label=metric_labels.get(metric, None),
                    )
                ax.set_title(n, loc='left')

            fig.supxlabel('Epoch')
            fig.legend(*ax.get_legend_handles_labels())

            return fig

    class EvaluationResult(dict):
        pass

    @staticmethod
    def _fetch_batch(
        ds: tf.data.Dataset,
        batch_size: int
    ):
        return (
            ds.ragged_batch(batch_size=batch_size)
            .map(lambda x: (x['frameseq'], x['label']))
            .prefetch(tf.data.AUTOTUNE)
        )

    @staticmethod
    def _infer_batch_size(ds_builder, scale: int = 1):
```

```
        return (
            ds_builder.info.features['label'].num_classes * scale
        )

    def __init__(self, model):
        self.model = model

    def train(
        self,
        ds_builder,
        batch_size: int | None = None,
        epochs: int | None = None,
        callbacks: typing.Sequence = [],
        **kwargs,
    ):
        return self.TrainingResult(
            cross_validate(
                self.model,
                ds_builder,
                splitter=KFoldSplitter(subset='train', n_splits=3),
                batch_fn=lambda ds: self._fetch_batch(
                    ds,
                    batch_size=batch_size if batch_size is not None
                        else self._infer_batch_size(ds_builder, scale↩
                            ↪=3),
                ),
                epochs=epochs if epochs is not None else 10,
                callbacks=[
                    keras.callbacks.EarlyStopping(
                        monitor='val_loss', patience=3,
                    ),
                    keras.callbacks.ProgbarLogger(),
                    *callbacks,
                ],
                **kwargs,
            )
        )

    def evaluate(
        self,
        ds_builder,
        batch_size: int | None = None,
        **kwargs,
    ):
        result = self.model.evaluate(
            self._fetch_batch(
                ds_builder.as_dataset(split='val'),
                batch_size=batch_size if batch_size is not None
                    else self._infer_batch_size(ds_builder),
            ),
            **kwargs,
        )
        return self.EvaluationResult(
            zip(self.model.metrics_names, result)
        )
```

```
model = Model(
    n_classes=(
        ds_builder.info
        .features['label']
        .num_classes
    )
)
manager = ModelManager(model)
```

A.5.3 Training

```
results_train = manager.train(ds_builder)
```

Demo: Result

```
%%space demo:clf:train

# loss
fig_loss = results_train.plot(
    fig=plt.figure(),
    metrics=['loss', 'val_loss'],
)
fig_loss.show()
scrapbook.glue(
    'figure:clf:train:loss',
    fig_loss,
)

# accuracy
fig_acc = results_train.plot(
    fig=plt.figure(),
    metrics=['accuracy', 'val_accuracy'],
)
fig_acc.show()
scrapbook.glue(
    'figure:clf:train:acc',
    fig_acc,
)
```

A.5.4 Evaluation

```
results_val = manager.evaluate(ds_builder)
scrapbook.glue(
    'data:clf:val',
    results_val,
)
results_val
```

A.6 Classification with Augmentations

A.6.1 Augmentations (Enhancements)

Histogram Equalization

```
import keras_cv

def Augmentation():
    return keras.layers.TimeDistributed(
        keras_cv.layers.Equalization((0., 255.))
    )
```

Demo: Equalization: Before, After

```
%%space demo:clf_augment:augment:histeq

class ImageHistViz:
    import numpy as np
    import matplotlib.pyplot as plt

    def __init__(self, im, cspace=None, ax=None):
        self.im = im
        self.cspace = cspace
        self.ax = ax or self.plt.gca()

    def plot(
        self,
        bins=256,
        cumulative=False,
        labels=None, colors=None
    ):
        cspace_info = {
            'rgb': {'labels': 'RGB', 'colors': 'rgb'}
        }

        shape_of = self.np.shape
        reshape = self.np.reshape

        for channel, color, label in zip(
            range(shape_of(self.im)[-1]),
            colors or cspace_info[self.cspace]['colors'],
            labels or cspace_info[self.cspace]['labels'],
        ):
            self.ax.hist(
                reshape(self.im[..., channel], (-1, )),
                bins=bins,
                cumulative=cumulative,
                histtype='step' if cumulative else 'stepfilled',
                color=color,
                alpha=1. / shape_of(self.im)[-1],
                label=label or channel,
            )
            self.ax.ticklabel_format(
                axis='y',
                style='scientific',
                scilimits=(0, 0),
            )

        return self

    def xylabel(self):
        self.ax.set(
            xlabel='Intensity',
            ylabel='Count',
        )
        return self

    def legend(self, *args, **kwargs):
        self.ax.legend(*args, **kwargs)
        return self
```

```
%%space demo:clf_augment:augment:histeq

class ImageHistVizGallery:
    import matplotlib.pyplot as plt

    @classmethod
```

```
    def stacked(cls, im, fig=None, subplot_kwds=dict(), **kwargs):
        fig = fig or cls.plt.gcf()
        return cls(
            im,
            axes=fig.subplots(
                nrows=2, ncols=1,
                sharex=True, sharey=False,
                **subplot_kwds,
            ).flat,
            **kwargs,
        )

    def __init__(self, im, axes, **histviz_kwds):
        self.im = im
        self.viz_hist = {
            key: ImageHistViz(im, ax=ax, **histviz_kwds)
            for key, ax in zip(['noncum', 'cum'], axes)
        }

    def plot(self):
        self.viz_hist['noncum'].plot(cumulative=False)
        self.viz_hist['cum'].plot(cumulative=True)
        return self

    def xylabel(self):
        for viz in self.viz_hist.values():
            viz.xylabel()
        return self

    def legend(self, *args, **kwargs):
        for viz in self.viz_hist.values():
            viz.legend(*args, **kwargs)
        return self
```

```
%%space demo:clf_augment:augment:histeq

imageseq = take_one(ds['train'])['frameseq']
def _imageseq_plot_before_after(fig=None):
    if fig is None:
        fig = plt.gca()

    subfigs = fig.subfigures(nrows=1, ncols=2)
    for (im, title), subfig in zip(
        [
            (imageseq[0].numpy(), 'Before'),
            (Augmentation()(imageseq)[0].numpy().astype(int), 'After')
        ],
        subfigs.flat,
    ):
        yield (im, title), subfig

# images
fig_im = plt.figure()
for (im, title), subfig in _imageseq_plot_before_after(fig_im):
    ax = subfig.subplots()
    ax.axis('off')
    ax.imshow(im)
fig_im.show()
scrapbook.glue(
    'figure:clf_augment:augment:histeq:im',
    fig_im,
    encoder='pickle'
)
```

```
# image histograms
fig_imhist = plt.figure()
for (im, title), subfig in _imageseq_plot_before_after(fig_imhist):
    (
        ImageHistVizGallery.stacked(
            im, cspace='rgb',
            fig=subfig,
        )
        .plot()
        .xylabel()
    )
fig_imhist.show()
scrapbook.glue(
    'figure:clf_augment:augment:histeq:imhist',
    fig_imhist,
    encoder='pickle'
)
```

A.6.2 Model

```
model_w_aug = Model(
    n_classes=(
        ds_builder.info
        .features['label']
        .num_classes
    ),
    augmentations=[Augmentation()],
)
manager_w_aug = ModelManager(model_w_aug)
```

A.6.3 Training

```
results_train_w_aug = manager_w_aug.train(ds_builder)
```

Demo: Result

```
%%space demo:clf_augment:train

# loss
fig_loss = results_train_w_aug.plot(
    fig=plt.figure(),
    metrics=['loss', 'val_loss'],
)
fig_loss.show()
scrapbook.glue(
    'figure:clf_augment:train:loss',
    fig_loss,
)

# accuracy
fig_acc = results_train_w_aug.plot(
    fig=plt.figure(),
    metrics=['accuracy', 'val_accuracy'],
)
fig_acc.show()
scrapbook.glue(
    'figure:clf_augment:train:acc',
    fig_acc,
)
```

A.6.4 Evaluation

```
results_val_w_aug = manager_w_aug.evaluate(ds_builder)
scrapbook.glue(
    'data:clf_augment:val',
    results_val_w_aug,
)
results_val_w_aug
```

A.6.5 Demo: Final Model

```
%%space demo:clf_augment:model

model_w_aug.summary()
```

```
%%space demo:clf_augment:model

dotfig_model_w_aug = keras.utils.model_to_dot(
    model_w_aug,
    show_shapes=True,
    show_dtype=False,
    show_layer_names=True,
    rankdir='TB',
    expand_nested=True,
    show_layer_activations=True,
    show_trainable=True,
)

ipy.core.display.SVG(dotfig_model_w_aug.create_svg())
scrapbook.glue(
    'data:clf_augment:dotfig_model',
    dotfig_model_w_aug,
    encoder='pickle'
)
```