# Appendix

## *A. Complete Code of Case Study 1*

```
import time

def format_duration(duration):
    """
    Format the duration expressed in seconds as
        string

    Parameters
    ----------
    duration : float
        a duration in seconds
    Return
    ------
    formated : str
        the given duration formated

    Example
    -------
    >>> days = 1
    >>> hours = 4
    >>> minutes = 52
    >>> seconds = 23
    >>> duration = seconds + 60*(minutes + 60*(hours
         + 24*days))
    >>> format_duration(duration)
    '1d 4h 52m 23s'
    """
    sec = duration % 60
    excess = int(duration) // 60  # minutes
    res = str(sec) + "s"
    if excess == 0:
        return res
    minutes = excess % 60
    excess = excess // 60  # hours
    res = str(minutes) + "m " + res
    if excess == 0:
        return res
    hour = excess % 24
    excess = excess // 24  # days
    res = str(hour) + "h " + res
    if excess == 0:
        return res
    res = str(excess)+"d " + res
    return res


def format_size(nb_bytes):
    """
    Format a size expressed in bytes as a string

    Parameters
    ----------
    nb_bytes : int
        the number of bytes

    Return
    ------
    formated : str
        the given number of bytes fromated

    Example
    -------
    >>> format_size(100)
    '100.0 bytes'
    >>> format_size(1000)
    '1.0 kB'
    >>> format_size(1000000)
    '1.0 MB'
    >>> format_size(1000000000)
    '1.0 GB'
    >>> format_size(1000000000000)
    '1.0 TB'
    >>> format_size(1000000000000000000)
    '1000000.0 TB'

    """
    for x in ['bytes', 'kB', 'MB', 'GB']:
        if nb_bytes < 1000.0:
            return "%3.1f %s" % (nb_bytes, x)
        nb_bytes /= 1000.0
    return "%3.1f %s" % (nb_bytes, 'TB')


class ProgressableTask:
    """
    ================
    ProgressableTask
    ================
    A :class:`ProgressableTask` is a task whose
        progression can n steps.
    Once the progression hits the max, the task
        status goes
    from `ProgressableTask.RUNNING` to `
        ProgressableTask.DONE`

    Note
    ----
    A task with 10 steps comprises 11 states : 0 (
        nothing is done yet)
    1-9 and 10 (task completed). In order to
        complete a task of 10 steps,
    the update must reach "10", that is, all the 10
        steps have been completed

    Class attributes
    ----------------
    nb_tasks : int
        The number of already created tasks. Beware
            that this is not
        thread-safe

    Class constants
    ---------------
    RUNNING : int
        State of a running task
    DONE : int
        State of a completed task

    Instance attributes
    -------------------
    id : int
        The id of the task
    name : str
        The name of the task

    Constructor parameters
    ----------------------
    nb_steps : int
        The number of step before completion
    name : str
        The name of the task
    """
    nb_tasks = 0

    RUNNING = 1
    DONE = 2
    UNFINISHED = 3

    def __init__(self, nb_steps, name=None):


        self.id = ProgressableTask.nb_tasks
        ProgressableTask.nb_tasks += 1

        if name is None:
            name = "Unnamed_task."+str(self.id)
        self.name = name

        self._nb_steps = nb_steps
        self._progress = 0
        self._last_reset = 0
        self._status = ProgressableTask.RUNNING
        self._end_time = None
        self._start_time = time.time()

    def get_nb_steps(self):
        """
        Returns
        -------
        nb_steps : int
            The number of steps required by the task
        """
        return self._nb_steps

    def __len__(self):
        steps =  self.get_nb_steps()
        if steps is None:
            raise AttributeError("Unbounded task")
        return steps

    def update(self, progress):
        """
        Update progress (if task is running)

        Parameters
        ----------
        progress : int
            the new progression score
        Return
        ------
        done : boolean
            True if the task is completed, False
                otherwise
        """
        if self._status == ProgressableTask.DONE:
            return True
        self._progress = progress
        if self._nb_steps is None:
            return False
        if progress >= self._nb_steps:
            self._progress = self._nb_steps
            self.close()
            return True
        else:
            return False

    def increment(self, inc=1):
        """
        Increment the progress by a given number

        Parameters
        ----------
        inc : int (Default : 1)
            The progress increment

        Return
        ------
        done : boolean
            True if the task is completed, False
                otherwise
        """
        return self.update(self.get_progress()+inc)

    def close(self, finished=True):
        """
        Ends the task

        Parameters
        ----------
        finished : boolean (Default : True)
            Whether the task has finished its
                excution correctly
        """
        if finished:
            self._status = ProgressableTask.DONE
        else:
            self._status = ProgressableTask.
                UNFINISHED
        self._end_time = time.time()

    def reset(self):
        """
        Reset the last progress counter. See method
            :`get_last_progress`
        """
        self._last_reset = self._progress

    def get_last_progress(self):
        """
        Return
        ------
        progress_ratio : float
            The relative progress since the last
                reset
        """
        if self._nb_steps is None:
            return 0
        return (float(self._progress - self.
            _last_reset)/self._nb_steps)


    def duration(self):
        """
        Return
        ------
        duration : float
            the duration of taks in seconds (up to
                now if still running,
            up to completion if completed)
        """
        if self._status == ProgressableTask.DONE:
            return self._end_time - self._start_time
        else:
            return time.time() - self._start_time

    def get_progress(self):
        return self._progress
```

```
    def is_completed(self):
        return self._status == ProgressableTask.DONE


class Formater:
    """
    ========
    Formater
    ========
    A :class:`Formater` format progresses as string

    Constructor parameters
    ----------------------
    format_func : callable (Default : str)
        A function to format progresses
    """

    def __init__(self, format_func=str,
        format_duration=format_duration):
        self._format_duration = format_duration
        self._format_func = format_func

    def format_task(self, task):
        return "Task " + str(task.id) +" '" + task.
            name + "' : "

    def format_progress(self, task):
        progress = task.get_progress()
        try:
            nb_steps = len(task)
            return self._format_func(progress)+"/"+
                self._format_func(nb_steps)
        except TypeError, AttributeError:
            return self._format_func(progress)



    def _format_nb_steps(self, task):
        str_nb_steps = "????"
        try:
            str_nb_steps = self._format_func(len(
                task))
        except TypeError, AttributeError:
            pass
        return str_nb_steps

    def format_creation(self, task):
        str_nb_steps = self._format_nb_steps(task)
        logging_message =  (self.format_task(task) +
             "Creation " + " (" + str_nb_steps + "
            steps)")
        return logging_message


    def format_end(self, task):
        duration = self._format_duration(task.
            duration())
        str_nb_steps = self._format_nb_steps(task)
        if task.is_completed():
            logging_msg = (self.format_task(task)+"
                Completion of the " +str_nb_steps +"
                 step(s) in " + duration)
        else:
            logging_msg = (self.format_task(task)+"
                Interruption after " +self.
                format_progress(task)+" step(s) in "
                 + duration)
        return logging_msg


def discard(msg, *args, **kwargs):
    return


class CompositeGenerator():

    def __init__(self, generator, inc=1):
        self.generator = generator
        self.inc = inc

    def __len__(self):
        return len(self.generator)

    def __iter__(self):
        for elem in self.generator:
            yield self.inc, elem
```

```
def log_iteration(composite_generator, name=None,
    log_func=discard,
                  formater=Formater(), log_ratio
                      =0.01):
    length = None
    try:
        length = len(composite_generator)
    except TypeError, AttributeError:
        pass
    task = ProgressableTask(length, name)

    try:
        # Log the start of the task
        log_func(formater.format_creation(task))
        # Running the decorated generator
        for inc, elem in composite_generator:
            # log the iterations of the task
            if not task.increment(inc):
                # Task is still in progress
                # Logging the message if necessary
                perc_prog = task.get_last_progress()
                if perc_prog >= log_ratio:
                    task.reset()
                    log_func(formater.
                        format_progress(task))
            # Yield the element
            yield elem
        task.close(True)
    except:
        task.close(False)
    finally:
        # Log the end of the task
        log_func(formater.format_end(task))


def log_loop(generator, name=None, log_func=discard,
         formater=Formater(), log_ratio=0.01):
    return log_iteration(CompositeGenerator(
        generator), name, log_func,
                         formater, log_ratio)


def log_transfer(generator, chunck_size, name=None,
    log_func=discard,
                 formater=Formater(format_func=
                     format_size), log_ratio=0.01):
    return log_iteration(CompositeGenerator(
        generator, chunck_size), name,
                         log_func, formater,
                             log_ratio)
```

### B. Complete Code of Case Study 2

```
import torch
...


def parse_charades_csv(filename):
    labels = {}
    with open(filename) as f:
        reader = csv.DictReader(f)
        for row in reader:
            vid = row['id']
            actions = row['actions']
            if actions == '':
                actions = []
            else:
                actions = [a.split(' ') for a in
                    actions.split(';')]
                actions = [{'class': x, 'start':
                    float(
                    y), 'end': float(z)} for x, y, z
                         in actions]
            labels[vid] = actions
    return labels


def cls2int(x):
    return int(x[1:])


def pil_loader(path):
    # open path as file to avoid ResourceWarning (
        https://github.com/python-pillow/Pillow/
        issues/835)
    with open(path, 'rb') as f:
        img = Image.open(f)
        return img.convert('RGB')


def accimage_loader(path):
    import accimage
    try:
        return accimage.Image(path)
    except IOError:
        # Potentially a decoding problem, fall back
          to PIL.Image
        return pil_loader(path)


def default_loader(path):
    from torchvision import get_image_backend
    if get_image_backend() == 'accimage':
        return accimage_loader(path)
    else:
        return pil_loader(path)


def cache(cachefile):
    """ Creates a decorator that caches the result
        to cachefile """
    def cachedecorator(fn):
        def newf(*args, **kwargs):
            print('cachefile {}'.format(cachefile))
            if os.path.exists(cachefile):
                with open(cachefile, 'rb') as f:
                    print("Loading cached result
                        from '%s'" % cachefile)
                    return pickle.load(f)
            res = fn(*args, **kwargs)
            with open(cachefile, 'wb') as f:
                print("Saving result to cache '%s'"
                    % cachefile)
                pickle.dump(res, f)
            return res
        return newf
    return cachedecorator


class Charades(data.Dataset):
    def __init__(self, root, split, labelpath,
        cachedir, transform=None, target_transform=
        None):
        self.num_classes = 157
        self.transform = transform
        self.target_transform = target_transform
        self.labels = parse_charades_csv(labelpath)
        self.root = root
        cachename = '{}/{}_{}.pkl'.format(cachedir,
            self.__class__.__name__, split)
        self.data = cache(cachename)(self.prepare)(
            root, self.labels, split)

    def prepare(self, path, labels, split):
        FPS, GAP, testGAP = 24, 4, 25
        datadir = path
        image_paths, targets, ids = [], [], []

        for i, (vid, label) in enumerate(labels.
            iteritems()):
            iddir = datadir + '/' + vid
            lines = glob(iddir+'/*.jpg')
            n = len(lines)
            if i % 100 == 0:
                print("{} {}".format(i, iddir))
            if n == 0:
                continue
            if split == 'val_video':
                target = torch.IntTensor(157).zero_
                    ()
                for x in label:
                    target[cls2int(x['class'])] = 1
                spacing = np.linspace(0, n-1,
                    testGAP)
                for loc in spacing:
                    impath = '{}/{}-{:06d}.jpg'.
                        format(
                        iddir, vid, int(np.floor(loc
                            ))+1)
                    image_paths.append(impath)
                    targets.append(target)
                    ids.append(vid)
            else:
                for x in label:
                    for ii in range(0, n-1, GAP):
                        if x['start'] < ii/float(FPS
                            ) < x['end']:
                            impath = '{}/{}-{:06d}.
                                jpg'.format(iddir,
                                vid, ii+1)
                            image_paths.append(
                                impath)
                            targets.append(cls2int(x
                                ['class']))
                            ids.append(vid)
        return {'image_paths': image_paths, 'targets
            ': targets, 'ids': ids}

    def __getitem__(self, index):
        """
        Args:
            index (int): Index
        Returns:
            tuple: (image, target) where target is
                class_index of the target class.
        """
        path = self.data['image_paths'][index]
        target = self.data['targets'][index]
        meta = {}
        meta['id'] = self.data['ids'][index]
        img = default_loader(path)
        if self.transform is not None:
            img = self.transform(img)
        if self.target_transform is not None:
            target = self.target_transform(target)
        return img, target, meta

    def __len__(self):
        return len(self.data['image_paths'])

    def __repr__(self):
        fmt_str = 'Dataset ' + self.__class__.
            __name__ + '\n'
        fmt_str += '    Number of datapoints: {}\n'.
            format(self.__len__())
        fmt_str += '    Root Location: {}\n'.format(
            self.root)
        tmp = '    Transforms (if any): '
        fmt_str += '{0}{1}\n'.format(
            tmp, self.transform.__repr__().replace('
                \n', '\n' + ' ' * len(tmp)))
        tmp = '    Target Transforms (if any): '
        fmt_str += '{0}{1}'.format(
            tmp, self.target_transform.__repr__().
                replace('\n', '\n' + ' ' * len(tmp))
                )
        return fmt_str


def get(args):
    """ Entry point. Call this function to get all
        Charades dataloaders """
    normalize = transforms.Normalize(mean=[0.485,
        0.456, 0.406], std=[0.229, 0.224, 0.225])
    train_file = args.train_file
    val_file = args.val_file
    train_dataset = Charades(
        args.data, 'train', train_file, args.cache,
        transform=transforms.Compose([
            transforms.RandomResizedCrop(args.
                inputsize),
            transforms.ColorJitter(
                brightness=0.4, contrast=0.4,
                    saturation=0.4),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),  # missing PCA
                lighting jitter
            normalize,
        ]))
    val_dataset = Charades(
        args.data, 'val', val_file, args.cache,
        transform=transforms.Compose([
            transforms.Resize(int(256./224*args.
                inputsize)),
            transforms.CenterCrop(args.inputsize),
            transforms.ToTensor(),
            normalize,
        ]))
    valvideo_dataset = Charades(
        args.data, 'val_video', val_file, args.cache
            ,
        transform=transforms.Compose([
            transforms.Resize(int(256./224*args.
                inputsize)),
            transforms.CenterCrop(args.inputsize),
            transforms.ToTensor(),
            normalize,
        ]))
```

```
    return train_dataset, val_dataset,
        valvideo_dataset
```