EventDance: Unsupervised Source-free Cross-modal Adaptation for Event-based Object Recognition
–Supplementray Material–

Xu Zheng[1] Lin Wang[1,2*]
[1]AI Thrust, HKUST(GZ) [2]Dept. of CSE, HKUST
zhengxu128@gmail.com, linwang@ust.hk
Project Page: https://vlislab22.github.io/EventDance/

Abstract

Due to the lack of space in the main paper, we provide more details of the proposed EventDance framework and experimental results in the supplementary material. Sec. 1 adds the Algorithm of the proposed DPPASS framework. Sec. 2 provides the implementation details of the proposed DPPASS method. Sec. 3 presents additional qualitative experimental results. Sec. 4 provides a detailed theoretical analysis of the unsupervised source-free cross-modal adaptation (USCA) problem. Finally, Sec. 5 provides code implementation of crucial modules.

1. Algorithm

The overall algorithm of EventDance is shown in Algorithm.1. Concretely, $\mathcal{L}_{EN}$ is used to optimize F_R; $\mathcal{L}_{TC}$ and $\mathcal{L}_{CM}$ are used to optimize F_S; and $\mathcal{L}_{Sup}$, $\mathcal{L}_{PC}$, and $\mathcal{L}_{CM}$ are used to optimize F_T^i. The whole framework is optimized in an end-to-end manner.

2. Implementation Details

In our source-free image-to-events adaptation setting, we aim to transfer knowledge from the pre-trained source model to learn event-based models in the target modality. Specifically, in our image-to-event scenario, we train the source model with gray-scale images which is transformed from the original RGB data in MINIST, CIFAR-10, and NCALTECH-101 dataset. In particular, EventDance employs three target models, taking three event representation types (stack image, voxel grid, and EST) as inputs in the training phase. Therefore, *the inference can be achieved using one of the models*.

We use ResNet-18, 34, and 50 (R-18, R-34, and R-50) pre-trained on ImageNet as backbones. Importantly, the

*Corresponding author.

Algorithm 1: Framework of our proposed method.

```
Input: Pre-trained source model: $F_S$;
Unlabeled events $X_T$; Reconstruction model $F_R$.
Output: Target models $F_T^i$, t ∈ {1,2,3}
Initialize $F_T^i$ randomly;
for each epoch do
    for each iteration do
        Sample a batch of target data $x_t$ from $X_T$;
        RMB module reconstructs surrogate images;
        Select anchor data $x_a$;
        Feed $x_a$ to $F_S$; Calculate Eq. 1;
        Calculate Eq. 2;
        Get pseudo labels $P = argmax(F_S(x_a))$;
        Supervise $F_T^i$ with $P$ as Eq. 3;
        Calculate Eq. 4;
        Calculate Eq. 5;
        Update parameters of $F_S$, $F_R$ and $F_T^i$.
    end
end
Final target models $F_T^i$, t ∈ $\{1, 2, 3\}$.
```

Property	CMKD [10]	ZDDA [16]	SOCKET [1]	Ours
Source Free	✗	✓	✓	✓
No Task-irrelevant data	✓	✗	✗	✓
Cross-modality	RGB $\rightarrow$ D	RGB$\rightarrow$ D	RGB$\rightarrow$ T/D	**I $\rightarrow$ E**
Unlabeled target data	✓	✗	✓	✓

Table 1. Comparison of our method with existing methods for knowledge transfer across different modalities. (D: depth; T: thermal; E: event; I: image)

source and target models use the same backbone in all the experiments in our EventDance. The batch size is set to 64, following the prior work [23]. We use the AdamW optimizer with a learning rate of 1e-5, which linearly decays over time. We use image augmentation techniques, *e.g.*, random rotations and flipping for source modality pre-

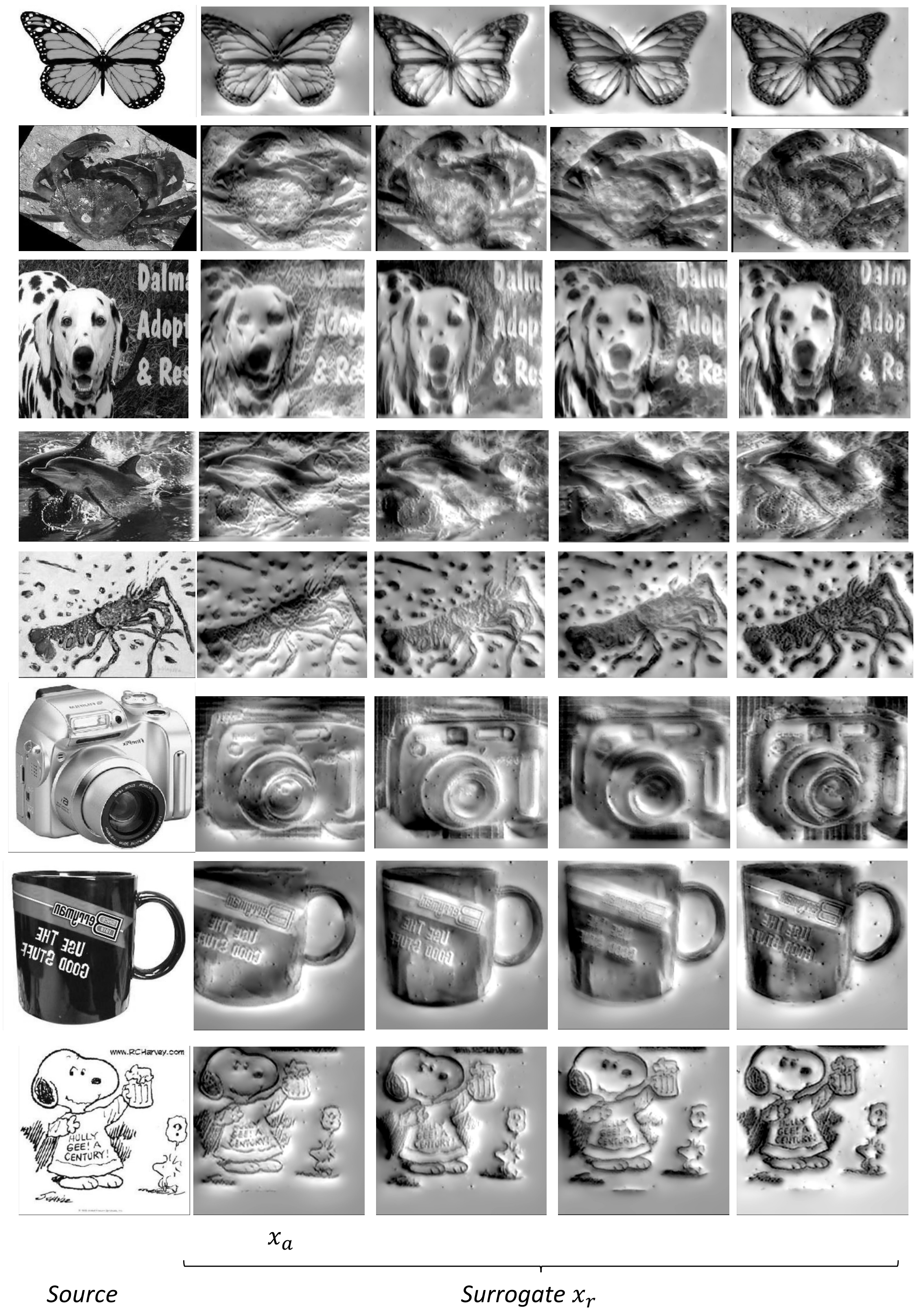

Figure 1. Visualization of samples in source and surrogate domain.

training. However, we do not use event augmentation techniques during target learning for a fair comparison with other methods. In our modality bridging module, we employ E2VID and Ev-FlowNet as our reconstruction model F_R and optical flow estimation model F_F, respectively. The E2VID model takes as input a voxel grid that passes through several layers, including a convolutional layer, three recurrent encoders that perform strided convolution followed by ConvLSTM, two residual blocks, three decoder layers that perform bilinear upsampling followed by convolution, and a final depthwise convolutional prediction layer. The model also includes skip connections between the symmetric encoder and decoder layers. The head layer has 32 output channels that double in number after each encoder. The head, encoder, and decoder layers use 5×5 kernels, while the remaining layers use 3×3 kernels. All layers, except for the final prediction layer, use ReLU activation functions. The final prediction layer uses a linear activation function. Ev-FlowNet takes the input voxel grid, Ek, and passes it through four strided convolutional layers with output channels doubling after each layer, starting from 64. The resulting activations are then processed by two residual blocks and four decoder layers that perform bilinear upsampling followed by convolution. Each decoder layer has a concatenated skip connection from the corresponding encoder and a depthwise convolution that produces a lower-scale flow estimate. This estimate is concatenated with the activations from the previous decoder layer. The LFlowNet loss (Eq. 10) is applied to each intermediate flow estimate via flow upsampling. All convolutional layers use 3×3 kernels and ReLU activation functions, except for the flow prediction layers, which use tanh activation functions [15].

3. More Experimental Comparison

Tab. 1 provides the comparison between our EventDance with existing methods for knowledge transfer across different modalities. We also provide the TSNE [21] visualization with more data samples in Fig. 2, apparently, our EventDance brings a significant improvement in distinguishing cross-modal samples in high-level feature space.

Meanwhile, we provide a performance comparison between our EventDance and supervised event-based recognition methods on the N-Caltech101 dataset in Tab. 2, including EV-VGCNN [8], MVF-Net [6], *etc*. Our EventDance achieves good performance in an unsupervised manner, without using the source data.

Method	S.F.	Unsup.	Backbone / Train	N-CAL
HFirst [14]	✗	✗	-	5.40
HOTS [11]	✗	✗	-	21.00
HATS [20]	✗	✗	-	64.20
EST [9]	✗	✗	-	81.70
RG-CNNs [3]	✗	✗	-	65.70
DART [17]	✗	✗	-	66.40
Matrix-LSTM [4]	✗	✗	-	84.30
ASCN [13]	✗	✗	-	74.50
EvS [12]	✗	✗	-	76.10
MVF-Net [6]	✗	✗	-	<u>87.10</u>
AEGNNs [18]	✗	✗	-	66.80
EV-VGCNN [7]	✗	✗	-	74.80
TORE [2]	✗	✗	-	79.80
E2VID [19]	✗	✓	Pre-train	64.00
E2VID [19]	✗	✓	Fine-tune	59.80
+ CLIP	✗	✓	Scratch	9.40
+ SSL	✗	✓	Pre-train	28.20
+ SSL	✗	✓	Scratch	30.50
Ev-LaFOR [5]	✗	✓	Text Prompt	82.46
+ CLIP	✗	✓	Visual Prompt	82.61
	✓	✓	-	42.70
Wang *et al.* [22]	✗	✓	-	43.50
+ CLIP	✓	✗	-	39.70
	✓	✓	R-18	66.77
EventDance (Ours)	✓	✓	R-34	72.68
	✓	✓	R-50	**92.35**

Table 2. Experimental results compared with more representative methods on event object recognition.

4. Theoretical Analysis

The principal objective of the training regimen is to facilitate the transfer of the source knowledge distribution, k_S, which is encapsulated by the source model F_S, and to apply it towards the learning of the target knowledge distribution, k_T, as embodied by the target model F_T^i. This process is visually delineated by the black arrows in Fig. 3.

However, as elucidated in the study by [1], the transition from a source modality to a target modality presents significantly more complexity than a mere domain shift across different datasets within the same modality. In response to this challenge, we propose the construction of an intermediary surrogate domain, denoted as X_R, which is designed to bridge the modality disparities through a reconstruction model F_R. This approach simplifies the Unsupervised Cross-modal Domain Adaptation by bifurcating the process into two distinct phases: (1) the extraction of knowledge from F_S, and (2) the subsequent indoctrination of F_T^i with the aforementioned extracted knowledge.

In the formulation of the surrogate domain $x_r \subseteq X_R$, we capitalize on the high temporal resolution afforded by raw events. By reconstructing multiple intensity frames from a stream of events and designating the initial reconstructed frames within x_r as the anchor data x_a, we establish a robust foundation for subsequent analysis (refer to Fig. 1). To enhance the efficacy of knowledge extraction, we invoke a

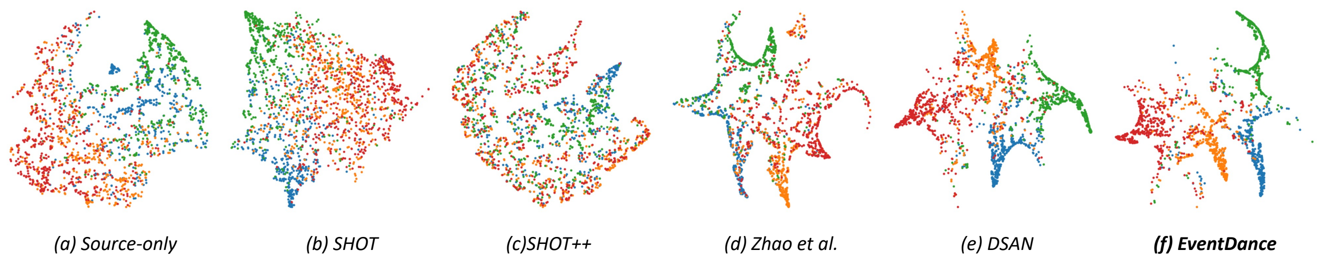

Figure 2. TSNE [21] visualization (with more data samples) of (a) source-only, (b) SHOT, (c) SHOT++, (d) Zhao *et al.*, (e)DSAN, and (f) EventDance on the target modality CIFAR10-DVS dataset with R-18 backbone. Different colors represent the 10 classes of CIFAR10-DVS dataset.

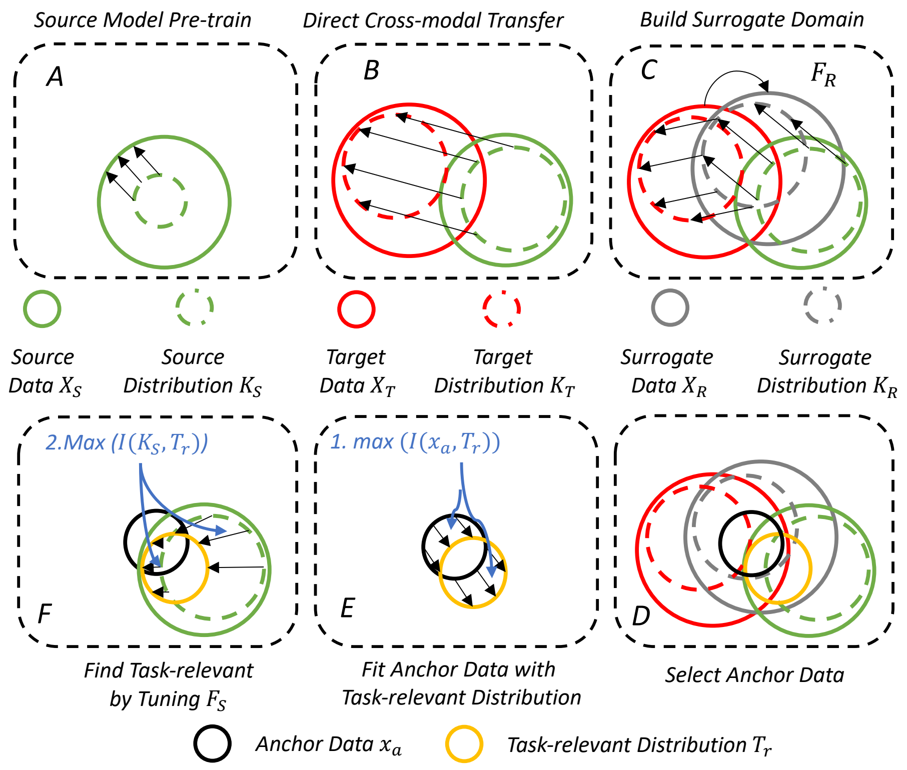
Source Model Pre-train
Direct Cross-modal Transfer
Build Surrogate Domain
A
B
C
F_R
Source Data X_S
Source Distribution K_S
Target Data X_T
Target Distribution K_T
Surrogate Data X_R
Surrogate Distribution K_R
2.Max $(I(K_S, T_r))$
1. max $(I(x_a, T_r))$
F
E
D
Find Task-relevant by Tuning F_S
Fit Anchor Data with Task-relevant Distribution
Select Anchor Data
Anchor Data x_a
Task-relevant Distribution T_r

Figure 3. High-level takeaways for our objectives of EventDance using information diagrams.

commonly held postulate:

Assumption 1: *A task-relevant knowledge distribution T_r is posited to encapsulate all the knowledge requisite for the final task, such as object recognition. This implies that if the distribution of x_a aligns with T_r, any knowledge extracted (in the form of pseudo labels) will be both reliable and congruent with the true labels.*

Conceptually, as depicted in Fig. 3 E, the alignment between the task-relevant knowledge distribution T_r and the surrogate domain x_r can be mathematically articulated as an optimization problem:

$$\max(MI(x_a; T_r)), \tag{1}$$

where $MI(\cdot)$ denotes the mutual information. The fundamental intent behind Eq. 1 is twofold: to excise information that is extraneous to the task and to amplify information that is pertinent, utilizing appropriate mechanisms. This duality of objectives can be re-envisioned in the following manner: given x_a's distribution within T_r, one must question the expected characteristics of x_a. Optimally, the response of F_S to x_a should exhibit a semblance to one-hot encoding. Moreover, as illustrated in Fig. 1, the set difference $x_r \setminus x_a$ may also contribute to the identification of the archetypal x_a, akin to a form of data augmentation. In light of this, Eq. 1 is decomposed into two objectives: (1) the minimization of the entropy $H(F_S(x_a))$, and (2) the maximization of the congruity between $F_S(x_a)$ and $F_S(x_r \setminus x_a)$.

Assumption 2: *Given that the source model F_S is pretrained on source data, it is presupposed to yield a proficient performance on the end task within the source modality. Consequently, it can be inferred that the knowledge distribution k_S shares a substantial intersection with the task-relevant distribution T_r.*

Based on Assumption 2, we postulate that the knowledge distribution k_S of the source model serves as a viable approximation of T_r, necessary for training purposes (refer to Fig. 3 F). This approach diverges from traditional Unsupervised Domain Adaptation (UDA) paradigms by promoting concurrent updates to both the source and target models within a reciprocal learning framework. Throughout the training process, k_S is iteratively refined to better align with T_r:

$$\max(I(k_S^{rgb}; T_r)). \tag{2}$$

Eq. 2 is concentrated on the iterative refinement of F_S to enhance its alignment with T_r. To this end, we introduce a mutual learning module that delicately coordinates the updates of both F_S and F_T^i, leveraging diverse event representations. In contradistinction to existing methodologies, which predominantly focus on event voxel grids as illustrated in [23], our strategy fully exploits the spatio-temporal dynamics intrinsic to raw events through multiple event representations, thereby tapping into the rich tapestry of information available within the event data.

5. Code Implementation

In this section, we provide some demo implementation codes of our proposed EventDance framework including event processing (*eventprocess.py*), training code of the events-to-image reconstruction (*reconstruction.py*), reconstruction net (*reconnet.py*), optical flow net (*flownet.py*), and classification model with event spike tensor (*EST.py*). The complete version will be publicly available upon acceptance.

```
import os
import numpy as np
import torch
import yaml
def events_to_image(xs, ys, ps, sensor_size=(180,
    240)):
    """
    Accumulate events into an image.
    """
    device = xs.device
    img_size = list(sensor_size)
    img = torch.zeros(img_size).to(device)
    if xs.dtype is not torch.long:
        xs = xs.long().to(device)
    if ys.dtype is not torch.long:
        ys = ys.long().to(device)
```

```
    img.index_put_((ys, xs), ps, accumulate=True)
    return img
def events_to_voxel(xs, ys, ts, ps, num_bins,
    sensor_size=(180, 240)):
    """
    Generate a voxel grid from input events using
     temporal bilinear interpolation.
    """
    assert len(xs) == len(ys) and len(ys) == len(
    ts) and len(ts) == len(ps)
    voxel = []
    ts = ts * (num_bins - 1)
    zeros = torch.zeros(ts.size())
    for b_idx in range(num_bins):
        weights = torch.max(zeros, 1.0 - torch.
    abs(ts - b_idx))
        voxel_bin = events_to_image(xs, ys, ps *
    weights, sensor_size=sensor_size)
        voxel.append(voxel_bin)
    return torch.stack(voxel)
```

eventprocess.py

```
import os
import argparse
import mlflow
import torch
from torch.optim import *
from configs.parser import YAMLParser
from dataloader.h5 import H5Loader
from loss.flow import EventWarping
from loss.reconstruction import
    BrightnessConstancy
from utils.utils import load_model,
    create_model_dir, save_model
from utils.visualization import Visualization
def train(args, config_parser):
    if not os.path.exists(args.path_models):
        os.makedirs(args.path_models)
    # configs
    config = config_parser.config
    config["vis"]["bars"] = False
    # log config
    mlflow.set_experiment(config["experiment"])
    mlflow.start_run()
    mlflow.log_params(config)
    mlflow.log_param("prev_model", args.
    prev_model)
    config["prev_model"] = args.prev_model
    # initialize settings
    device = config_parser.device
    kwargs = config_parser.loader_kwargs
    num_bins = config["data"]["num_bins"]
    # visualization tool
    if config["vis"]["enabled"]:
        vis = Visualization(config)
    # data loader
    data = H5Loader(config, num_bins)
    dataloader = torch.utils.data.DataLoader(
        data,
        drop_last=True,
        batch_size=config["loader"]["batch_size"
    ],
        collate_fn=data.custom_collate,
        worker_init_fn=config_parser.
    worker_init_fn,
        **kwargs
)
# loss functions
loss_function_flow = EventWarping(config,
device)
loss_function_reconstruction =
BrightnessConstancy(config, device)
# reconstruction settings
model_reconstruction = eval(config["
model_reconstruction"]["name"])(
    config["model_reconstruction"].copy(),
num_bins
).to(device)
model_reconstruction = load_model(args.
prev_model, model_reconstruction, device)
model_reconstruction.train()
# optical flow settings
model_flow = eval(config["model_flow"]["name"
])(config["model_flow"].copy(), num_bins).to(
device)
model_flow = load_model(args.prev_model,
model_flow, device)
if config["loss"]["train_flow"]:
    model_flow.train()
else:
    model_flow.eval()
# model directory
path_models = create_model_dir(args.
path_models, mlflow.active_run().info.run_id)
mlflow.log_param("trained_model", path_models
)
config_parser.log_config(path_models)
config["trained_model"] = path_models
config_parser.config = config
config_parser.log_config(path_models)
# optimizers
optimizer_reconstruction = eval(config["
optimizer"]["name"])(
    model_reconstruction.parameters(), lr=
config["optimizer"]["lr"]
)
optimizer_flow = eval(config["optimizer"]["
name"])(model_flow.parameters(), lr=config["
optimizer"]["lr"])
optimizer_reconstruction.zero_grad()
optimizer_flow.zero_grad()
# simulation variables
seq_length = 0
loss_reconstruction = 0
loss_flow = 0
train_loss_reconstruction = 0
train_loss_flow = 0
best_loss_reconstruction = 1.0e6
best_loss_flow = 1.0e6
end_train = False
prev_img = None
x_reconstruction = None
# training loop
data.shuffle()
while True:
    print(len(dataloader))
    for inputs in dataloader:
        if data.new_seq:
            seq_length = 0
            data.new_seq = False
            loss_reconstruction = 0
            model_reconstruction.reset_states
()
```

```
            optimizer_reconstruction.
zero_grad()
            prev_img = None
            x_reconstruction = None
        if data.seq_num >= len(data.files):
            mlflow.log_metric(
                "loss_reconstruction",
train_loss_reconstruction / (data.samples +
1), step=data.epoch
            )
            mlflow.log_metric("loss_flow",
train_loss_flow / (data.samples + 1), step=
data.epoch)
            with torch.no_grad():
                if train_loss_reconstruction
/ (data.samples + 1) <
best_loss_reconstruction:
                    save_model(path_models,
model_reconstruction)
                    best_loss_reconstruction
= train_loss_reconstruction / (data.samples +
 1)
                if train_loss_flow / (data.
samples + 1) < best_loss_flow:
                    save_model(path_models,
model_flow)
                    best_loss_flow =
train_loss_flow / (data.samples + 1)
            data.epoch += 1
            data.samples = 0
            train_loss_flow = 0
            train_loss_reconstruction = 0
            data.seq_num = data.seq_num % len
(data.files)
            # finish training loop
            if data.epoch == config["loader"
]["n_epochs"]:
                end_train = True
        # forward pass - flow network
        # inputs["inp_voxel"] [1,5,128,128]
inputs["inp_cnt"] [1,2,128,128]
        x_flow = model_flow(inputs["inp_voxel
"].to(device), inputs["inp_cnt"].to(device))
        # loss and backward pass
        if config["loss"]["train_flow"]:
            loss_flow = loss_function_flow(
                x_flow["flow"], inputs["
inp_list"].to(device), inputs["inp_pol_mask"
].to(device)
            )
            train_loss_flow += loss_flow.item
()
            loss_flow.backward()
            optimizer_flow.step()
            optimizer_flow.zero_grad()
        if x_reconstruction is not None:
            # reconstruction loss -
generative model
            delta_loss_model =
loss_function_reconstruction.generative_model
(
                x_flow["flow"][0].detach(),
x_reconstruction["image"], inputs
            )
            loss_reconstruction +=
delta_loss_model
            train_loss_reconstruction +=
delta_loss_model.item()
                if prev_img is None or "Pause"
not in data.batch_augmentation or not data.
batch_augmentation["Pause"]:
                    # reconstruction loss -
regularization
                    delta_loss_reg =
loss_function_reconstruction.regularization(
x_reconstruction["image"])
                    loss_reconstruction +=
delta_loss_reg
                    train_loss_reconstruction +=
delta_loss_reg.item()
                    # update previous image
                    prev_img = x_reconstruction["
image"].detach().clone()
            # forward pass - reconstruction
network
            x_reconstruction =
model_reconstruction(inputs["inp_voxel"].to(
device))
            data.tc_idx += 1
            # reconstruction loss - temporal
constancy
            if data.tc_idx >= config["loss"]["
reconstruction_tc_idx_threshold"]:
                delta_loss_tc =
loss_function_reconstruction.
temporal_consistency(
                    x_flow["flow"][0].detach(),
prev_img, x_reconstruction["image"]
                )
                loss_reconstruction +=
delta_loss_tc
                train_loss_reconstruction +=
delta_loss_tc.item()
            # update sequence length
            seq_length += 1
            # visualize
            with torch.no_grad():
                if config["vis"]["enabled"] and
config["loader"]["batch_size"] == 1:
                    vis.update(inputs, x_flow["
flow"][-1], None, x_reconstruction["image"])
            # reconstruction backward pass
            if seq_length == config["loss"]["
reconstruction_unroll"]:
                if loss_reconstruction != 0:
                    loss_reconstruction.backward
()
                    optimizer_reconstruction.step
()
                    optimizer_reconstruction.
zero_grad()
                seq_length = 0
                x_reconstruction = None
                loss_reconstruction = 0
                # detach states
                model_reconstruction.
detach_states()
            # print training info
            if config["vis"]["verbose"]:
                print(
                    "Train␣Epoch:␣{:04d}␣[{:03d
}/{:03d}␣({:03d}%)]␣Flow␣loss:␣{:.6f},␣
Brightness␣loss:␣{:.6f}".format(
                        data.epoch,
                        data.seq_num,
                        len(data.files),
                        int(100 * data.seq_num /
    len(data.files)),
                        train_loss_flow / (data.
    samples + 1),
                        train_loss_reconstruction
     / (data.samples + 1),
                    ),
                    end="\r",
                )
            # update number of samples seen by
    the network
            data.samples += config["loader"]["
    batch_size"]
        if end_train:
            break
    mlflow.end_run()
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "--config",
        default="configs/train_reconstruction.yml
    ",
        help="training␣configuration",
    )
    parser.add_argument(
        "--path_models",
        default="trained_models/",
        help="location␣of␣trained␣models",
    )
    args = parser.parse_args()
    # launch training
    train(args, YAMLParser(args.config))
```

reconstruction.py

```
import copy
import numpy as np
import torch
from .base import BaseModel
from .model_util import copy_states,
    CropParameters
from .submodules import ResidualBlock, ConvGRU,
    ConvLayer
from .unet import UNetRecurrent, MultiResUNet
class E2VID(BaseModel):
    """
    E2VID architecture for image reconstruction
    from event-data.
    "High speed and high dynamic range video with
     an event camera", Rebecq et al. 2019.
    """
    def __init__(self, unet_kwargs, num_bins):
        super().__init__()
        self.crop = None
        norm = None
        use_upsample_conv = True
        final_activation = "none"
        if "norm" in unet_kwargs.keys():
            norm = unet_kwargs["norm"]
        if "use_upsample_conv" in unet_kwargs.
    keys():
            use_upsample_conv = unet_kwargs["
    use_upsample_conv"]
        if "final_activation" in unet_kwargs.keys
    ():
            final_activation = unet_kwargs[”
    final_activation”]
        E2VID_kwargs = {
            ”base_num_channels”: unet_kwargs[”
    base_num_channels”],
            ”num_encoders”: 3,
            ”num_residual_blocks”: 2,
            ”num_output_channels”: 1,
            ”skip_type”: ”sum”,
            ”norm”: norm,
            ”num_bins”: num_bins,
            ”use_upsample_conv”:
    use_upsample_conv,
            ”kernel_size”: unet_kwargs[”
    kernel_size”],
            ”channel_multiplier”: 2,
            ”recurrent_block_type”: ”convlstm”,
            ”final_activation”: final_activation,
        }
        self.num_encoders = E2VID_kwargs[”
    num_encoders”]
        unet_kwargs.update(E2VID_kwargs)
        unet_kwargs.pop(”name”, None)
        unet_kwargs.pop(”encoding”, None)  # TODO
    : remove
        self.unetrecurrent = UNetRecurrent(
    unet_kwargs)
    @property
    def states(self):
        return copy_states(self.unetrecurrent.
    states)
    @states.setter
    def states(self, states):
        self.unetrecurrent.states = states
    def detach_states(self):
        detached_states = []
        for state in self.unetrecurrent.states:
            if type(state) is tuple:
                tmp = []
                for hidden in state:
                    tmp.append(hidden.detach())
                detached_states.append(tuple(tmp)
    )
            else:
                detached_states.append(state.
    detach())
        self.unetrecurrent.states =
    detached_states
    def reset_states(self):
        self.unetrecurrent.states = [None] * self
    .unetrecurrent.num_encoders
    def init_cropping(self, width, height,
    safety_margin=0):
        self.crop = CropParameters(width, height,
     self.num_encoders, safety_margin)
    def forward(self, inp_voxel):
        ”””
        :param inp_voxel: N x num_bins x H x W
        :return: [N x 1 X H X W] reconstructed
    brightness signal.
        ”””
        # pad input
        x = inp_voxel
        if self.crop is not None:
            x = self.crop.pad(x)
        # forward pass
        img = self.unetrecurrent.forward(x)
```

```
        # crop output
        if self.crop is not None:
            img = img[:, :, self.crop.iy0 : self.
    crop.iy1, self.crop.ix0 : self.crop.ix1]
            img = img.contiguous()
        return {”image”: img}
class FireNet(BaseModel):
    ”””
    FireNet architecture for image reconstruction
     from event-data.
    ”Fast image reconstruction with an event
    camera”, Scheerlinck et al., 2019
    ”””
    def __init__(self, unet_kwargs, num_bins):
        super().__init__()
        base_num_channels = unet_kwargs[”
    base_num_channels”]
        kernel_size = unet_kwargs[”kernel_size”]
        padding = kernel_size // 2
        self.head = ConvLayer(num_bins,
    base_num_channels, kernel_size, padding=
    padding)
        self.G1 = ConvGRU(base_num_channels,
    base_num_channels, kernel_size)
        self.R1 = ResidualBlock(base_num_channels
    , base_num_channels)
        self.G2 = ConvGRU(base_num_channels,
    base_num_channels, kernel_size)
        self.R2 = ResidualBlock(base_num_channels
    , base_num_channels)
        self.pred = ConvLayer(base_num_channels,
    out_channels=1, kernel_size=1, activation=
    None)
        self.num_encoders = 0  # needed by
    image_reconstructor.py
        self.num_recurrent_units = 2
        self.reset_states()
    @property
    def states(self):
        return copy_states(self._states)
    @states.setter
    def states(self, states):
        self._states = states
    def detach_states(self):
        detached_states = []
        for state in self.states:
            if type(state) is tuple:
                tmp = []
                for hidden in state:
                    tmp.append(hidden.detach())
                detached_states.append(tuple(tmp)
    )
            else:
                detached_states.append(state.
    detach())
        self.states = detached_states
    def reset_states(self):
        self._states = [None] * self.
    num_recurrent_units
    def init_cropping(self, width, height):
        pass
    def forward(self, inp_voxel):
        ”””
        :param inp_voxel: N x num_bins x H x W
        :return: [N x 1 X H X W] reconstructed
    brightness signal.
        ”””
        # forward pass
        x = inp_voxel
        x = self.head(x)
        x = self.G1(x, self._states[0])
        self._states[0] = x
        x = self.R1(x)
        x = self.G2(x, self._states[1])
        self._states[1] = x
        x = self.R2(x)
        return {"image": self.pred(x)}
```

reconnet.py

```
import copy
import numpy as np
import torch
from .base import BaseModel
from .model_util import copy_states,
    CropParameters
from .submodules import ResidualBlock, ConvGRU,
    ConvLayer
from .unet import UNetRecurrent, MultiResUNet
class EVFlowNet(BaseModel):
    """
    EV-FlowNet architecture for (dense/sparse)
    optical flow estimation from event-data.
    "EV-FlowNet: Self-Supervised Optical Flow for
     Event-based Cameras", Zhu et al. 2018.
    """
    def __init__(self, unet_kwargs, num_bins):
        super().__init__()
        self.crop = None
        self.mask = unet_kwargs["mask_output"]
        EVFlowNet_kwargs = {
            "base_num_channels": unet_kwargs["
    base_num_channels"],
            "num_encoders": 4,
            "num_residual_blocks": 2,
            "num_output_channels": 2,
            "skip_type": "concat",
            "norm": None,
            "num_bins": num_bins,
            "use_upsample_conv": True,
            "kernel_size": unet_kwargs["
    kernel_size"],
            "channel_multiplier": 2,
            "final_activation": "tanh",
        }
        self.num_encoders = EVFlowNet_kwargs["
    num_encoders"]
        unet_kwargs.update(EVFlowNet_kwargs)
        unet_kwargs.pop("name", None)
        unet_kwargs.pop("eval", None)
        unet_kwargs.pop("encoding", None)  # TODO
    : remove
        unet_kwargs.pop("mask_output", None)
        unet_kwargs.pop("mask_smoothing", None)
    # TODO: remove
        if "flow_scaling" in unet_kwargs.keys():
            unet_kwargs.pop("flow_scaling", None)
        self.multires_unet = MultiResUNet(
    unet_kwargs)
    def reset_states(self):
        pass
    def init_cropping(self, width, height,
    safety_margin=0):
        self.crop = CropParameters(width, height,
     self.num_encoders, safety_margin)
    def forward(self, inp_voxel, inp_cnt):
        """
        :param inp_voxel: N x num_bins x H x W
        :return: output dict with list of [N x 2
    X H X W] (x, y) displacement within
    event_tensor.
        """
        # pad input
        x = inp_voxel
        if self.crop is not None:
            x = self.crop.pad(x)
        # forward pass
        multires_flow = self.multires_unet.
    forward(x)
        # upsample flow estimates to the original
     input resolution
        flow_list = []
        for flow in multires_flow:
            flow_list.append(
                torch.nn.functional.interpolate(
                    flow,
                    scale_factor=(
                        multires_flow[-1].shape
    [2] / flow.shape[2],
                        multires_flow[-1].shape
    [3] / flow.shape[3],
                    ),
                )
            )
        # crop output
        if self.crop is not None:
            for i, flow in enumerate(flow_list):
                flow_list[i] = flow[:, :, self.
    crop.iy0 : self.crop.iy1, self.crop.ix0 :
    self.crop.ix1]
                flow_list[i] = flow_list[i].
    contiguous()
        # mask flow
        if self.mask:
            mask = torch.sum(inp_cnt, dim=1,
    keepdim=True)
            mask[mask > 0] = 1
            for i, flow in enumerate(flow_list):
                flow_list[i] = flow * mask
        return {"flow": flow_list}
class FireFlowNet(BaseModel):
    """
    FireFlowNet architecture for (dense/sparse)
    optical flow estimation from event-data.
    "Back to Event Basics: Self Supervised
    Learning of Image Reconstruction from Event
    Data via Photometric Constancy", Paredes-
    Valles et al., 2020
    """
    def __init__(self, unet_kwargs, num_bins):
        super().__init__()
        base_num_channels = unet_kwargs["
    base_num_channels"]
        kernel_size = unet_kwargs["kernel_size"]
        self.mask = unet_kwargs["mask_output"]
        padding = kernel_size // 2
        self.E1 = ConvLayer(num_bins,
    base_num_channels, kernel_size, padding=
    padding)
        self.E2 = ConvLayer(base_num_channels,
    base_num_channels, kernel_size, padding=
    padding)
        self.R1 = ResidualBlock(base_num_channels
    , base_num_channels)
        self.E3 = ConvLayer(base_num_channels,
    base_num_channels, kernel_size, padding=
    padding)
        self.R2 = ResidualBlock(base_num_channels
    , base_num_channels)
        self.pred = ConvLayer(base_num_channels,
    out_channels=2, kernel_size=1, activation="
    tanh")
    def reset_states(self):
        pass
    def init_cropping(self, width, height):
        pass
    def forward(self, inp_voxel, inp_cnt):
        """
        :param inp_voxel: N x num_bins x H x W
        :return: output dict with list of [N x 2
    X H X W] (x, y) displacement within
    event_tensor.
        """
        # forward pass
        x = inp_voxel
        x = self.E1(x)
        x = self.E2(x)
        x = self.R1(x)
        x = self.E3(x)
        x = self.R2(x)
        flow = self.pred(x)
        # mask flow
        if self.mask:
            mask = torch.sum(inp_cnt, dim=1,
    keepdim=True)
            mask[mask > 0] = 1
            flow = flow * mask
        return {"flow": [flow]}
```

flownet.py

```
import torch.nn as nn
from os.path import join, dirname, isfile
import torch
import torch.nn.functional as F
import numpy as np
from torchvision.models.resnet import resnet34,
    resnet18
import tqdm
class ValueLayer(nn.Module):
    def __init__(self, mlp_layers, activation=nn.
    ReLU(), num_channels=9):
        assert mlp_layers[-1] == 1, "Last␣layer␣
    of␣the␣mlp␣must␣have␣1␣input␣channel."
        assert mlp_layers[0] == 1, "First␣layer␣
    of␣the␣mlp␣must␣have␣1␣output␣channel"
        nn.Module.__init__(self)
        self.mlp = nn.ModuleList()
        self.activation = activation
        # create mlp
        in_channels = 1
        for out_channels in mlp_layers[1:]:
            self.mlp.append(nn.Linear(in_channels
    , out_channels))
            in_channels = out_channels
        # init with trilinear kernel
        path = join(dirname(__file__), "
    quantization_layer_init", "trilinear_init.pth
    ")
        if isfile(path):
            state_dict = torch.load(path)
            self.load_state_dict(state_dict)
        else:
            self.init_kernel(num_channels)
    def forward(self, x):
        # create sample of batchsize 1 and input
    channels 1
        x = x[None,...,None]
        # apply mlp convolution
        for i in range(len(self.mlp[:-1])):
            x = self.activation(self.mlp[i](x))
        x = self.mlp[-1](x)
        x = x.squeeze()
        return x
    def init_kernel(self, num_channels):
        ts = torch.zeros((1, 2000))
        optim = torch.optim.Adam(self.parameters
    (), lr=1e-2)
        torch.manual_seed(1)
        for _ in tqdm.tqdm(range(1000)):  #
    converges in a reasonable time
            optim.zero_grad()
            ts.uniform_(-1, 1)
            # gt
            gt_values = self.trilinear_kernel(ts,
     num_channels)
            # pred
            values = self.forward(ts)
            # optimize
            loss = (values - gt_values).pow(2).
    sum()
            loss.backward()
            optim.step()
    def trilinear_kernel(self, ts, num_channels):
        gt_values = torch.zeros_like(ts)
        gt_values[ts > 0] = (1 - (num_channels-1)
     * ts)[ts > 0]
        gt_values[ts < 0] = ((num_channels-1) *
    ts + 1)[ts < 0]
        gt_values[ts < -1.0 / (num_channels-1)] =
     0
        gt_values[ts > 1.0 / (num_channels-1)] =
    0
        return gt_values
class QuantizationLayer(nn.Module):
    def __init__(self, dim,
                 mlp_layers=[1, 100, 100, 1],
                 activation=nn.LeakyReLU(
    negative_slope=0.1)):
        nn.Module.__init__(self)
        self.value_layer = ValueLayer(mlp_layers,
                                      activation=
    activation,

    num_channels=dim[0])
        self.dim = dim
    def forward(self, events):
        # points is a list, since events can have
     any size
        B = int((1+events[-1,-1]).item())
        num_voxels = int(2 * np.prod(self.dim) *
    B)
        vox = events[0].new_full([num_voxels,],
    fill_value=0)
        C, H, W = self.dim
        # get values for each channel
        x, y, t, p, b = events.t()
        # normalizing timestamps
        for bi in range(B):
            t[events[:,-1] == bi] /= t[events
    [:,-1] == bi].max()
        p = (p+1)/2  # maps polarity to 0, 1
        idx_before_bins = x \
                          + W * y \
                          + 0 \
                          + W * H * C * p \
                          + W * H * C * 2 * b
        for i_bin in range(C):
            values = t * self.value_layer.forward
    (t-i_bin/(C-1))
            # draw in voxel grid
            idx = idx_before_bins + W * H * i_bin
            vox.put_(idx.long(), values,
    accumulate=True)
        vox = vox.view(-1, 2, C, H, W)
        vox = torch.cat([vox[:, 0, ...], vox[:,
    1, ...]], 1)
        return vox
class Classifier(nn.Module):
    def __init__(self,
                 voxel_dimension=(9,180,240),  #
    dimension of voxel will be C x 2 x H x W
                 crop_dimension=(224, 224),  #
    dimension of crop before it goes into
    classifier
                 num_classes=101,
                 mlp_layers=[1, 30, 30, 1],
                 activation=nn.LeakyReLU(
    negative_slope=0.1),
                 pretrained=True):
        nn.Module.__init__(self)
        self.quantization_layer =
    QuantizationLayer(voxel_dimension, mlp_layers
    , activation)
        self.classifier = resnet18(pretrained=
    pretrained)
        self.crop_dimension = crop_dimension
        # replace fc layer and first
    convolutional layer
        input_channels = 2*voxel_dimension[0]
        self.classifier.conv1 = nn.Conv2d(
    input_channels, 64, kernel_size=7, stride=2,
    padding=3, bias=False)
        self.classifier.fc = nn.Linear(self.
    classifier.fc.in_features, num_classes)
    def crop_and_resize_to_resolution(self, x,
    output_resolution=(224, 224)):
        B, C, H, W = x.shape
        if H > W:
            h = H // 2
            x = x[:, :, h - W // 2:h + W // 2, :]
        else:
            h = W // 2
            x = x[:, :, :, h - H // 2:h + H // 2]
        x = F.interpolate(x, size=
    output_resolution)
        return x
    def forward(self, x):
        vox = self.quantization_layer.forward(x)
        vox_cropped = self.
    crop_and_resize_to_resolution(vox, self.
    crop_dimension)
        pred = self.classifier.forward(
    vox_cropped)
        return pred, vox
```

EST.py

References

[1] Sk Miraj Ahmed, Suhas Lohit, Kuan-Chuan Peng, Michael Jones, and Amit K. Roy-Chowdhury. Cross-modal knowledge transfer without task-relevant source data. In *ECCV*, pages 111–127. Springer, 2022. 1, 3

[2] Raymond Baldwin, Ruixu Liu, Mohammed Mutlaq Almatrafi, Vijayan K Asari, and Keigo Hirakawa. Time-ordered recent event (tore) volumes for event cameras. *IEEE TPAMI*, 2022. 3

[3] Yin Bi, Aaron Chadha, Alhabib Abbas, Eirina Bourtsoulatze, and Yiannis Andreopoulos. Graph-based spatio-temporal feature learning for neuromorphic vision sensing. *TIP*, 29: 9084–9098, 2020. 3

[4] Marco Cannici, Marco Ciccone, Andrea Romanoni, and Matteo Matteucci. A differentiable recurrent surface for asynchronous event-based data. In *ECCV*, pages 136–152. Springer, 2020. 3

[5] Hoonhee Cho, Hyeonseong Kim, Yujeong Chae, and Kuk-Jin Yoon. Label-free event-based object recognition via joint learning with image reconstruction from events. *arXiv preprint arXiv:2308.09383*, 2023. 3

[6] Yongjian Deng, Hao Chen, and Youfu Li. Mvf-net: A multi-view fusion network for event-based object classification. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(12):8275–8284, 2021. 3

[7] Yongjian Deng, Hao Chen, Hai Liu, and Youfu Li. A voxel graph cnn for object classification with event cameras. In *CVPR*, pages 1172–1181, 2022. 3

[8] Yongjian Deng, Hao Chen, Hai Liu, and Youfu Li. A voxel graph cnn for object classification with event cameras. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1172–1181, 2022. 3

[9] Daniel Gehrig, Antonio Loquercio, Konstantinos G Derpanis, and Davide Scaramuzza. End-to-end learning of representations for asynchronous event-based data. In *ICCV*, pages 5633–5643, 2019. 3

[10] Saurabh Gupta, Judy Hoffman, and Jitendra Malik. Cross modal distillation for supervision transfer. In *CVPR*, pages 2827–2836, 2016. 1

[11] Xavier Lagorce, Garrick Orchard, Francesco Galluppi, Bertram E Shi, and Ryad B Benosman. Hots: a hierarchy of event-based time-surfaces for pattern recognition. *IEEE TPAMI*, 39(7):1346–1359, 2016. 3

[12] Yijin Li, Han Zhou, Bangbang Yang, Ye Zhang, Zhaopeng Cui, Hujun Bao, and Guofeng Zhang. Graph-based asynchronous event processing for rapid object recognition. In *ICCV*, pages 934–943, 2021. 3

[13] Nico Messikommer, Daniel Gehrig, Antonio Loquercio, and Davide Scaramuzza. Event-based asynchronous sparse convolutional networks. In *ECCV*, 2020. 3

[14] Garrick Orchard, Cedric Meyer, Ralph Etienne-Cummings, Christoph Posch, Nitish Thakor, and Ryad Benosman. Hfirst: A temporal approach to object recognition. *IEEE transactions on pattern analysis and machine intelligence*, 37(10): 2028–2040, 2015. 3

[15] Federico Paredes-Vallés and Guido CHE de Croon. Back to event basics: Self-supervised learning of image reconstruction for event cameras via photometric constancy. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3446–3455, 2021. 3

[16] Kuan-Chuan Peng, Ziyan Wu, and Jan Ernst. Zero-shot deep domain adaptation. In *ECCV*, pages 764–781, 2018. 1

[17] Bharath Ramesh, Hong Yang, Garrick Orchard, Ngoc Anh Le Thi, Shihao Zhang, and Cheng Xiang. Dart: distribution aware retinal transform for event-based cameras. *TPAMI*, 42 (11):2767–2780, 2019. 3

[18] Simon Schaefer, Daniel Gehrig, and Davide Scaramuzza. Aegnn: Asynchronous event-based graph neural networks. In *CVPR*, pages 12371–12381, 2022. 3

[19] Cedric Scheerlinck, Henri Rebecq, Daniel Gehrig, Nick Barnes, Robert Mahony, and Davide Scaramuzza. Fast image reconstruction with an event camera. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 156–163, 2020. 3

[20] Amos Sironi, Manuele Brambilla, Nicolas Bourdis, Xavier Lagorce, and Ryad Benosman. Hats: Histograms of averaged time surfaces for robust event-based object classification. In *CVPR*, pages 1731–1740, 2018. 3

[21] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9 (11), 2008. 3, 4

[22] Lin Wang, Yo-Sung Ho, Kuk-Jin Yoon, et al. Event-based high dynamic range image and very high frame rate video generation using conditional generative adversarial networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10081–10090, 2019. 3

[23] Junwei Zhao, Shiliang Zhang, and Tiejun Huang. Transformer-based domain adaptation for event data classification. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2022, Virtual and Singapore, 23-27 May 2022*, pages 4673–4677. IEEE, 2022. 1, 5