Fire Diff

Release 1.0.0

Christopher Ondrusz

Aug 30, 2024

CONTENTS:

CHAPTER

ONE

MODELS

1.1 fireDiff.Models.unet

class fireDiff.Models.unet.**Bottleneck**(*embed_dim=128*, *num_heads=8*)

Bases: Module

A bottleneck layer with a convolutional block and multi-head attention, followed by a 1x1 convolution to reduce the number of channels.

1.1.1 Parameters:

embed_dim
[int, optional, default=128] The dimensionality of the embeddings used in the multi-head attention.

num_heads
[int, optional, default=8] The number of heads in the multi-head attention mechanism.

1.1.2 Returns:

torch.Tensor
The output tensor after the bottleneck, with 128 channels and the same spatial dimensions as the input.

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Forward pass through the bottleneck layer.

Parameters:

x
[torch.Tensor] The input tensor of shape (batch_size, 64, height, width).

Returns:

torch.Tensor
The output tensor of shape (batch_size, 128, height, width).

class fireDiff.Models.unet.**ConvBlock**(*in_channels*, *out_channels*)

Bases: Module

A convolutional block that applies two convolutional layers with batch normalization and activation functions.

1.1.3 Parameters:

in_channels
[int] The number of input channels for the convolutional layers.

out_channels
[int] The number of output channels for the convolutional layers.

1.1.4 Returns:

torch.Tensor
The output tensor after applying the convolutional block, with the same height and width as the input.

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Forward pass through the convolutional block.

Parameters:

x
[torch.Tensor] The input tensor of shape (batch_size, in_channels, height, width).

Returns:

torch.Tensor

The output tensor of shape (batch_size, out_channels,
height, width).

class fireDiff.Models.unet.**Decoder**

Bases: Module

The decoder part of a U-Net, which upsamples the input using transposed convolutions and refines the features using convolutional blocks.

1.1.5 Returns:

torch.Tensor
The final output tensor after the decoder, with 1 channel and the same spatial dimensions as the original input.

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*, *enc_features*)

Forward pass through the decoder.

Parameters:

x
: [torch.Tensor] The input tensor from the bottleneck layer of shape (batch_size, 128, height/8, width/8).

enc_features
: [tuple] A tuple of feature maps from the encoder to be concatenated with the upsampled features at each step.

Returns:

torch.Tensor
: The output tensor of shape (batch_size, 1, height, width) after applying the final GELU activation.

`class fireDiff.Models.unet.Encoder`

Bases: `Module`

The encoder part of a U-Net, which reduces the spatial dimensions of the input through convolutional blocks and max-pooling layers.

1.1.6 Returns:

tuple :
: A tuple containing: - The downsampled output tensor after the final pooling layer. - A tuple of feature maps from each convolutional block, to be used in the decoder for skip connections.

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward(x)`

Forward pass through the encoder.

Parameters:

x
: [torch.Tensor] The input tensor of shape (batch_size, 1, height, width).

Returns:

tuple :
: A tuple containing: - The output tensor after the final pooling layer, of shape (batch_size, 64, height/8, width/8). - A tuple of tensors containing the output of each convolutional block before pooling, to be used in the decoder for skip connections.

`class fireDiff.Models.unet.UNet`

Bases: `Module`

A U-Net model combining an encoder, a bottleneck with an attention mechanism, and a decoder to produce a final output with the same spatial dimensions as the input.

1.1.7 Returns:

torch.Tensor
The final output tensor after passing through the U-Net, with 1 channel and the same spatial dimensions as the input.

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward(x)`

Forward pass through the U-Net model.

Parameters:

x
[torch.Tensor] The input tensor of shape (batch_size, 1, height, width).

Returns:

torch.Tensor
The output tensor of shape (batch_size, 1, height, width) after passing through the entire U-Net.

1.2 fireDiff.Models.unet_predictor

`class fireDiff.Models.unet_predictor.Bottleneck(embed_dim=128, num_heads=8)`

Bases: `Module`

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward(x)`

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`class fireDiff.Models.unet_predictor.ConvBlock(in_channels, out_channels)`

Bases: `Module`

A convolutional block consisting of two convolutional layers, each followed by a ReLU activation. The block preserves the spatial dimensions of the input.

1.2.1 Parameters:

in_channels
[int] The number of input channels for the convolutional layers.

out_channels
[int] The number of output channels for the convolutional layers.

1.2.2 Returns:

None

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class fireDiff.Models.unet_predictor.**Decoder**

Bases: `Module`

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*, *enc_features*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class fireDiff.Models.unet_predictor.**Encoder**

Bases: `Module`

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class fireDiff.Models.unet_predictor.**PredictorUNet**

Bases: `Module`

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

1.3 fireDiff.Models.diffusionmodel

class fireDiff.Models.diffusionmodel.**DiffusionModel**(*model*, *lambda_start=1.0*, *lambda_end=0.0*, *device='cpu'*)

Bases: `Module`

A diffusion model that uses a base neural network model (e.g., UNet) to perform denoising and sampling tasks. The model can be trained using a specified noise schedule.

1.3.1 Parameters:

model
[nn.Module] The base neural network model architecture (e.g., UNet).

lambda_start
[float, optional, default=1.0] The starting value for the lambda schedule.

lambda_end
[float, optional, default=0.0] The ending value for the lambda schedule.

device
[str, optional, default='cuda' if torch.cuda.is_available() else 'cpu' # noqa] The device to run the model on ('cuda' or 'cpu').

1.3.2 Returns:

None

Initialize internal Module state, shared by both nn.Module and ScriptModule.

diffusion_sampler(*x_t*, *num_steps=50*)

Generates samples from the diffusion model.

Parameters:

x_t
[torch.Tensor] The input tensor representing the noisy data to be denoised.

num_steps
[int, optional, default=50] The number of sampling steps to perform.

Returns:

torch.Tensor
The denoised output tensor generated by the model.

`load_model`(*path*)

Loads the model's state dictionary from a specified path.

Parameters:

path
[str] The file path from which to load the model's state dictionary.

Returns:

None

`save_model`(*path*)

Saves the model's state dictionary to a specified path.

Parameters:

path
[str] The file path where the model's state dictionary will be saved.

Returns:

None

`train_model`(*dataloader*, *train_steps=1000*, *epochs=50*)

Trains the diffusion model using the provided dataloader.

Parameters:

dataloader
[torch.utils.data.DataLoader] The dataloader providing batches of data for training.

train_steps
[int, optional, default=1000] The number of diffusion steps used in training.

epochs
[int, optional, default=50] The number of epochs for which to train the model.

Returns:

None

`class fireDiff.Models.deterministicmodel.`**`PredictionModel`**(*model*, *device='cpu'*)

Bases: `Module`

A wrapper class for a neural network model that facilitates training, validation, and saving/loading model checkpoints. The model is designed to run on either a CPU or GPU.

1.3.3 Parameters:

model
[nn.Module] The base model architecture (e.g., UNet).

device
[str, optional, default='cuda' if torch.cuda.is_available() else 'cpu' # noqa] The device to run the model on ('cuda' or 'cpu').

1.3.4 Returns:

None

`load_model`(*path*)

Loads the model's state dictionary from a specified file.

Parameters:

path
[str] The file path from which to load the model's state dictionary.

Returns:

None

`save_model`(*path*)

Saves the model's state dictionary to a specified file.

Parameters:

path
[str] The file path where the model's state dictionary will be saved.

Returns:

None

`train_model`(*tdataloader*, *vdataloader*, *epochs=50*)

Trains the model over a specified number of epochs, and evaluates it on a validation dataset after each epoch.

Parameters:

tdataloader
[torch.utils.data.DataLoader] The data loader providing the training data.

vdataloader
[torch.utils.data.DataLoader] The data loader providing the validation data.

epochs
[int, optional, default=50] The number of epochs to train the model.

Returns:

tuple
A tuple containing two lists: the training loss and validation loss for each epoch.

`train_step`(*optimizer*, *criterion*, *data_loader*)

Performs a single training step, including forward pass, loss computation, and backpropagation for the entire dataset provided by the data loader.

Parameters:

optimizer
[torch.optim.Optimizer] The optimizer used to update the model's weights.

criterion
[nn.Module] The loss function used to compute the loss.

data_loader
[torch.utils.data.DataLoader] The data loader providing the training data.

Returns:

float
The average training loss over the entire dataset.

`validate`(*criterion*, *data_loader*)

Evaluates the model on a validation dataset without updating model weights.

Parameters:

criterion
[nn.Module] The loss function used to compute the loss.

data_loader
[torch.utils.data.DataLoader] The data loader providing the validation data.

Returns:

float
The average validation loss over the entire dataset.

1.4 fireDiff.Models.cae

`class fireDiff.Models.cae.Autoencoder`

Bases: `Module`

A convolutional autoencoder neural network for image compression and reconstruction. The autoencoder consists of an encoder that compresses the input image into a lower-dimensional representation and a decoder that reconstructs the image from this representation.

1.4.1 Parameters:

None

1.4.2 Returns:

None

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Defines the forward pass of the autoencoder, where the input image is passed through the encoder to obtain a compressed representation and then through the decoder to reconstruct the image.

Parameters:

x
[torch.Tensor] The input tensor of shape (batch_size, 1, height, width) representing grayscale images.

Returns:

torch.Tensor
The reconstructed image tensor of the same shape as the input.

1.5 fireDiff.Models.fcae

class fireDiff.Models.fcae.**FullyConnectedAutoencoder**

Bases: Module

A fully connected autoencoder neural network for image compression and reconstruction. The autoencoder compresses the input image into a lower-dimensional representation using fully connected layers and then reconstructs the image from this representation.

1.5.1 Parameters:

None

1.5.2 Returns:

None

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Defines the forward pass of the fully connected autoencoder. The input image is first flattened, then passed through the encoder to obtain a compressed representation, and finally passed through the decoder to reconstruct the image.

Parameters:

x
: [torch.Tensor] The input tensor of shape (batch_size, 1, 128, 128) representing grayscale images.

Returns:

torch.Tensor
: The reconstructed image tensor of shape (batch_size, 1, 128, 128).

1.6 fireDiff.Models.utils

fireDiff.Models.utils.**compute_alpha_sigma**(*lambda_tau*)

Compute the alpha and sigma values used in the diffusion process from the given lambda values.

1.6.1 Parameters:

lambda_tau
: [torch.Tensor] A tensor containing lambda values, which are used to calculate the alpha and sigma values.

1.6.2 Returns:

tuple of torch.Tensor
: A tuple containing two tensors: - *alpha_tau*: The calculated alpha values, which are used in the diffusion process. - *sigma_tau*: The calculated sigma values, representing the standard deviation of the noise added during diffusion.

1.6.3 Example:

```
>>> lambda_tau = torch.tensor([0.5, 1.0, 1.5])
>>> alpha_tau, sigma_tau = compute_alpha_sigma(lambda_tau)
>>> print(alpha_tau)
>>> print(sigma_tau)
```

fireDiff.Models.utils.**loss_function**(*v_tau*, *v_hat*, *lambda_tau*, *t*)

Compute the loss for training a diffusion model, considering the predicted and true noise.

1.6.4 Parameters:

v_tau
: [torch.Tensor] The true noise tensor used in the diffusion process.

v_hat
: [torch.Tensor] The predicted noise tensor by the model.

lambda_tau
: [torch.Tensor] A tensor containing lambda values for each timestep.

t
: [torch.Tensor] A tensor of timesteps at which the loss is computed.

1.6.5 Returns:

torch.Tensor
The computed loss value, which is used to optimize the diffusion model during training.

1.6.6 Example:

```
>>> v_tau = torch.randn(1, 3, 32, 32)
>>> v_hat = torch.randn(1, 3, 32, 32)
>>> lambda_tau = torch.linspace(0.1, 0.9, 50)
>>> t = torch.tensor([10])
>>> loss = loss_function(v_tau, v_hat, lambda_tau, t)
>>> print(loss.item())
```

fireDiff.Models.utils.**noise_schedule**(*timesteps*, *schedule_type='linear'*)

Generate a noise schedule based on the specified type and number of timesteps.

1.6.7 Parameters:

timesteps
[int] The number of timesteps for which to generate the noise schedule.

schedule_type
[str, optional] The type of noise schedule to generate. Supported types are 'linear' and 'cosine'. Default is 'linear'.

1.6.8 Returns:

torch.Tensor
A tensor containing the noise schedule values for each timestep.

1.6.9 Raises:

ValueError
If the *schedule_type* is not 'linear' or 'cosine'.

1.6.10 Example:

```
>>> timesteps = 50
>>> schedule = noise_schedule(timesteps, schedule_type='cosine')
>>> print(schedule)
```

fireDiff.Models.utils.**surrogate_target**(*x_t*, *t*, *alpha*, *sigma*)

Calculate the surrogate target used in the training of diffusion models.

1.6.11 Parameters:

x_t
[torch.Tensor] The diffused tensor at timestep *t*.

t
[int] The current timestep.

alpha
[torch.Tensor] A tensor containing alpha values for each timestep.

sigma
[torch.Tensor] A tensor containing sigma values for each timestep.

1.6.12 Returns:

torch.Tensor
The surrogate target tensor, which is used for model training to approximate the noise added during diffusion.

1.6.13 Example:

```
>>> x_t = torch.randn(1, 3, 32, 32)
>>> t = 10
>>> alpha = torch.linspace(0.1, 0.9, 50)
>>> sigma = torch.linspace(0.9, 0.1, 50)
>>> target = surrogate_target(x_t, t, alpha, sigma)
>>> print(target.shape)
```

fireDiff.Models.utils.**variance_preserving_diffusion**(*x_0*, *t*, *alpha_tau*, *sigma_tau*)
Apply variance-preserving diffusion to an input tensor at a specific timestep.

1.6.14 Parameters:

x_0
[torch.Tensor] The original input tensor, representing the initial data (e.g., an image).

t
[int] The timestep at which to apply the diffusion process.

alpha_tau
[torch.Tensor] A tensor containing precomputed alpha values for each timestep.

sigma_tau
[torch.Tensor] A tensor containing precomputed sigma values for each timestep.

1.6.15 Returns:

torch.Tensor
The diffused tensor *z_t*, which is the result of applying the diffusion process to *x_0* at timestep *t*.

1.6.16 Example:

```
>>> x_0 = torch.randn(1, 3, 32, 32)
>>> t = 10
>>> alpha_tau = torch.linspace(0.1, 0.9, 50)
>>> sigma_tau = torch.linspace(0.9, 0.1, 50)
>>> z_t = variance_preserving_diffusion(x_0, t, alpha_tau, sigma_tau)
>>> print(z_t.shape)
```

CHAPTER

TWO

DATASETS

2.1 fireDiff.Datasets.pairdataset

class fireDiff.Datasets.pairdataset.**VideoFramePairsDataset**(*video_dir*, *transform=None*, *timegap=1*)

Bases: Dataset

A dataset class for creating pairs of grayscale video frames from a directory of video files.

2.1.1 Attributes:

video_dir
[str] The directory containing the video files.

transform
[callable, optional] A function/transform to apply to the video frames.

timegap
[int, optional] The gap between consecutive frames in a pair, default is 1.

video_files
[list of str] A list of paths to the video files in the directory.

frame_pairs
[list of tuples] A list of tuples, where each tuple contains two consecutive frames with a time gap.

Initializes the VideoFramePairsDataset.

2.1.2 Parameters:

video_dir
[str] The directory containing the video files.

transform
[callable, optional] A function/transform to apply to the video frames.

timegap
[int, optional] The gap between consecutive frames in a pair, default is 1.

2.2 fireDiff.Datasets.videodataset

class fireDiff.Datasets.videodataset.**VideoDataset**(*video_folder*, *num_videos*, *num_frames_per_video*, *frame_size*, *transform=None*)

Bases: Dataset

A PyTorch Dataset class for loading frames from multiple videos stored in a folder. The frames are converted to grayscale and can be optionally transformed.

2.2.1 Parameters:

video_folder
[str] The directory containing the video files.

num_videos
[int] The number of videos to load from the folder.

num_frames_per_video
[int] The number of frames to load from each video.

frame_size
[tuple] The desired frame size (height, width) for the loaded frames.

transform
[callable, optional, default=None] A function/transform that takes in a frame and returns a transformed version.

2.2.2 Returns:

None

load_video(*video_path*)

Loads a single video from the specified path, converts its frames to grayscale, and resizes them to the specified frame size.

Parameters:

video_path
[str] The path to the video file to be loaded.

Returns:

np.ndarray
An array of grayscale frames from the video.

load_videos()

Loads frames from multiple videos in the specified folder. Each video is loaded, converted to grayscale, resized, and a specified number of frames are selected.

Returns:

list
A list containing all the loaded and processed frames from all videos.

CHAPTER

THREE

UTILITIES

3.1 fireDiff.Utils.utilities

fireDiff.Utils.utilities.**calculate_matching_percentage**(*tensor1*, *tensor2*)

Calculate the percentage of elements that have the same value between two tensors.

3.1.1 Parameters:

tensor1
: [torch.Tensor] The first input tensor. This tensor can have any shape, but it must match the shape of *tensor2*.

tensor2
: [torch.Tensor] The second input tensor, which must have the same shape as *tensor1*.

3.1.2 Returns:

float
: The percentage of elements that match between the two tensors. This is calculated as the number of matching elements divided by the total number of elements, multiplied by 100.

3.1.3 Raises:

ValueError
: If the input tensors do not have the same shape.

3.1.4 Example:

```
>>> import torch
>>> tensor1 = torch.tensor([[1, 2, 3], [4, 5, 6]])
>>> tensor2 = torch.tensor([[1, 2, 0], [4, 5, 6]])
>>> percentage = calculate_matching_percentage(tensor1, tensor2)
>>> print(f"Percentage of matching elements: {percentage}%")
Percentage of matching elements: 83.33333333333334%
```

fireDiff.Utils.utilities.**threshold**(*image*, *value=0*)

Apply a threshold to an image tensor, converting all pixel values to either 1 or -1 based on the threshold value.

3.1.5 Parameters:

image
: [torch.Tensor] The input image as a PyTorch tensor. This tensor can have any shape, but typically it will be a 2D or 3D tensor representing an image.

value
: [int or float, optional] The threshold value to apply. Any pixel value in the input image greater than or equal to this value will be set to 1, and any pixel value below this value will be set to -1. The default threshold value is 0.

3.1.6 Returns:

torch.Tensor
: A tensor with the same shape as the input image, where each pixel is either 1 or -1 depending on the thresholding.

3.1.7 Example:

```
>>> import torch
>>> image = torch.tensor([[0.5, -0.2], [1.2, 0.0]])
>>> thresholded_image = threshold(image, value=0.1)
>>> print(thresholded_image)
(tensor([[ 1, -1],
         [ 1, -1]]),)
```

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

N

P

S

T

U

V