Arbitrary Style Transfer with Adaptive Instance Normalization (AdaIN)

Abstract

This report presents a comprehensive analysis of the pytorch-AdaIN implementation, which is an unofficial PyTorch implementation of the paper "Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization" by Huang and Belongie (ICCV 2017). The project enables real-time neural style transfer by combining the content of one image with the artistic style of another image using a novel normalization technique called Adaptive Instance Normalization (AdaIN).

1. Introduction

Neural style transfer is a technique that applies the artistic style of one image to the content of another image. Traditional methods like Gatys et al. (2016) require iterative optimization for each image pair, making them computationally expensive. The AdaIN approach addresses this limitation by using a feed-forward network that can perform style transfer in real-time while maintaining high-quality results.

2. Model Architecture and Principles

2.1 Adaptive Instance Normalization (AdaIN)

The core innovation of this model is the Adaptive Instance Normalization (AdaIN) operation. AdaIN transfers the style of one image to another by aligning the mean and standard deviation of the content features with those of the style features.

Mathematical Formulation:

Given content features `c` and style features `s`, AdaIN is defined as:

```
AdaIN(c, s) = σ(s) * ((c - μ(c)) / σ(c)) + μ(s)
```

Where:

- `μ(c)` and `σ(c)` are the mean and standard deviation of content features
- `μ(s)` and `σ(s)` are the mean and standard deviation of style features

This operation normalizes the content features to have zero mean and unit variance, then scales and shifts them to match the style feature statistics. This effectively transfers the style while preserving the spatial structure of the content.

2.2 Network Architecture

The model consists of three main components:

2.2.1 Encoder (VGG-19)

- **Purpose**: Extracts feature representations from input images
- **Architecture**: Pre-trained VGG-19 network (up to relu4_1 layer)
- **Frozen Parameters**: The encoder weights are fixed during training
- **Output**: 512-channel feature maps at relu4_1 layer

The encoder extracts features at multiple levels:

- `relu1_1` : Low-level features (edges, textures)
- `relu2_1` : Mid-level features
- `relu3_1` : Higher-level features
- `relu4_1` : High-level semantic features (used for AdaIN)

2.2.2 AdaIN Layer

- **Location**: Between encoder and decoder
- **Function**: Applies adaptive instance normalization to align content and style statistics
- **Implementation**: Located in `function.py` as `adaptive_instance_normalization()`

2.2.3 Decoder

- **Purpose**: Reconstructs the stylized image from normalized features
- **Architecture**: Symmetric decoder with upsampling layers
- **Structure**:
  - Input: 512 channels (from relu4_1)
  - Progressive upsampling: 512 → 256 → 128 → 64 → 3 channels
  - Uses reflection padding and nearest-neighbor upsampling
  - Only the decoder is trained during training

2.3 Training Process

The model is trained using a combination of content loss and style loss:

Content Loss:

```
L_content = ||f(g_t) - t||²
```

Where `g_t` is the decoder output and `t` is the AdaIN-transformed feature.

Style Loss:

```
L_style = Σ_i ||μ(f_i(g_t)) - μ(f_i(s))||² + ||σ(f_i(g_t)) - σ(f_i(s))||²
```

Where the sum is over multiple feature layers (relu1_1, relu2_1, relu3_1, relu4_1).

Total Loss:

```
L_total = λ_c * L_content + λ_s * L_style
```

With default weights: `λ_c = 1.0` and `λ_s = 10.0`

Training Details:

- Only the decoder is trained (encoder is frozen)
- Optimizer: Adam with learning rate 1e-4
- Learning rate decay: 5e-5
- Batch size: 8
- Maximum iterations: 160,000
- Image size: 256×256 (random crop from 512×512)

3. Code Structure

3.1 File Organization

```
pytorch-AdaIN/
├── net.py               # Network architecture definitions
├── function.py          # Core algorithms (AdaIN, color preservation)
├── test.py             # Inference script for image style transfer
├── train.py            # Training script
├── test_video.py       # Video style transfer script
├── sampler.py          # Infinite sampler for training
├── models/             # Pre-trained model files
│   ├── decoder.pth
│   └── vgg_normalised.pth
└── requirements.txt    # Python dependencies
```

3.2 Key Components

3.2.1 `net.py`

- `vgg` : VGG-19 encoder network (up to relu4_1)
- `decoder` : Symmetric decoder network
- `Net` **class**: Main network class that combines encoder and decoder
  - `encode()` : Extracts relu4_1 features
  - `encode_with_intermediate()` : Extracts features at multiple levels
  - `calc_content_loss()` : Computes content loss
  - `calc_style_loss()` : Computes style loss
  - `forward()` : Forward pass for training

3.2.2 `function.py`

- `calc_mean_std()` : Calculates mean and standard deviation of feature maps
- `adaptive_instance_normalization()` : Core AdaIN implementation
- `coral()` : Color preservation algorithm (CORAL: Correlation Alignment)

3.2.3 `test.py`

- **Main inference script** for single image or batch processing
- **Key features**:
  - Single image style transfer
  - Batch processing with directory inputs
  - Style interpolation (multiple styles)
  - Color preservation option
  - Adjustable style strength (alpha parameter)
  - Default paths: `my_content/` and `my_style/`

3.2.4 `train.py`

- **Training script** with the following components:
  - `FlatFolderDataset` : Dataset class for loading images
  - `InfiniteSamplerWrapper` : Infinite sampling for training
  - `adjust_learning_rate()` : Learning rate scheduling
  - Training loop with loss computation and optimization

3.3 Data Flow

Inference Flow:

1. Load content and style images
2. Preprocess images (resize, normalize)
3. Extract features using VGG encoder:
   - Content → `content_feat` (relu4_1)
   - Style → `style_feat` (relu4_1)
4. Apply AdaIN: `t = AdaIN(content_feat, style_feat)`
5. Blend with content (alpha parameter): `t = alpha * t + (1-alpha) * content_feat`
6. Decode to image: `output = decoder(t)`
7. Save result

Training Flow:

1. Load random content and style image pairs
2. Forward pass through encoder → AdaIN → decoder
3. Compute content and style losses at multiple layers
4. Backpropagate (only through decoder)
5. Update decoder weights
6. Repeat for 160,000 iterations

4. Usage Guide

4.1 Installation

1. **Install dependencies:**

```
pip install -r requirements.txt
```

Required packages:

- Python 3.5+
- PyTorch 0.4+
- TorchVision
- Pillow
- NumPy
- OpenCV (for video processing)
- TensorBoardX (for training visualization)

1. **Download pre-trained models:**
   Download from: https://github.com/naoto0804/pytorch-AdaIN/releases/tag/v0.0.0

- `decoder.pth` : Trained decoder weights
- `vgg_normalised.pth` : VGG-19 encoder weights

Place both files in the `models/` directory.

4.2 Basic Usage

Single Image Style Transfer

Method 1: Using default directories (simplest)

```
# Place your images in:
# - my_content/ (content images)
# - my_style/ (style images)

python test.py
```

Method 2: Specify image paths

```
python test.py --content <content_image_path> --style <style_image_path>
```

Example:

```
python test.py --content input/content/cornell.jpg --style input/style/woman_with_hat_matisse.jpg
```

Batch Processing

Process all combinations of images in two directories:

```
python test.py --content_dir <content_dir> --style_dir <style_dir>
```

Note: If `content_dir` has N images and `style_dir` has M images, this will generate N×M output images (all possible combinations).

Example:

```
python test.py --content_dir my_content --style_dir my_style
```

4.3 Advanced Options

Style Strength Control (`--alpha`)

Controls the degree of stylization (0.0 to 1.0):

```
# Strong stylization (default)
python test.py --content photo.jpg --style style.jpg --alpha 1.0

# Medium strength (recommended for portraits)
python test.py --content photo.jpg --style style.jpg --alpha 0.5

# Light stylization
python test.py --content photo.jpg --style style.jpg --alpha 0.3
```

Color Preservation (`--preserve_color`)

Preserves the original color palette of the content image:

```
python test.py --content photo.jpg --style style.jpg --preserve_color
```

Image Size Control

```
# High quality (slower, requires GPU)
python test.py --content photo.jpg --style style.jpg --content_size 1024

# Default size
python test.py --content photo.jpg --style style.jpg --content_size 512

# Fast processing
python test.py --content photo.jpg --style style.jpg --content_size 256
```

Output Directory

```
python test.py --content photo.jpg --style style.jpg --output my_results
```

Style Interpolation

Blend multiple styles:

```
python test.py --content photo.jpg \
  --style style1.jpg,style2.jpg,style3.jpg \
  --style_interpolation_weights 1,1,1
```

4.4 Training

To train your own decoder:

```
python train.py --content_dir <content_dir> --style_dir <style_dir>
```

Training parameters:

- `--lr` : Learning rate (default: 1e-4)
- `--max_iter` : Maximum iterations (default: 160000)
- `--batch_size` : Batch size (default: 8)
- `--style_weight` : Style loss weight (default: 10.0)
- `--content_weight` : Content loss weight (default: 1.0)

4.5 Video Style Transfer

Apply style transfer to video:

```
python test_video.py --content_video <video_path> --style_path <style_image_or_video>
```

5. Technical Details

5.1 AdaIN Implementation

The AdaIN operation is implemented in `function.py` :

```
def adaptive_instance_normalization(content_feat, style_feat):
    style_mean, style_std = calc_mean_std(style_feat)
    content_mean, content_std = calc_mean_std(content_feat)

    normalized_feat = (content_feat - content_mean) / content_std
    return normalized_feat * style_std + style_mean
```

5.2 Feature Extraction

The encoder extracts features at the `relu4_1` layer (31st layer of VGG-19), which provides a good balance between:

- **Content preservation**: High enough to capture semantic structure
- **Style transfer**: Low enough to allow style manipulation

5.3 Decoder Architecture

The decoder uses:

- **Reflection padding**: Prevents border artifacts
- **Nearest-neighbor upsampling**: Preserves feature values
- **Progressive channel reduction**: 512 → 256 → 128 → 64 → 3

5.4 Color Preservation

The CORAL (Correlation Alignment) algorithm is used for color preservation:

- Aligns the color distribution of the style image to match the content image
- Preserves the original color palette while applying style textures

6. Advantages and Limitations

6.1 Advantages

1. **Real-time performance**: Feed-forward network enables fast inference
2. **Arbitrary style transfer**: Works with any style image without retraining
3. **Controllable stylization**: Alpha parameter allows fine-tuning
4. **High-quality results**: Produces visually appealing stylized images
5. **Flexible usage**: Supports single images, batches, and videos

6.2 Limitations

1. **Fixed encoder**: Uses pre-trained VGG-19, limiting flexibility
2. **Single scale**: Processes images at a single resolution level
3. **Style-content trade-off**: Balancing content preservation and style transfer can be challenging
4. **Training data dependency**: Decoder quality depends on training data diversity

7. Experimental Results

The model successfully transfers artistic styles from various paintings (Van Gogh, Picasso, Matisse, etc.) to photographs while preserving the content structure. Key observations:

- **Alpha parameter**: Lower values (0.3-0.5) work better for portraits, higher values (0.8-1.0) for landscapes
- **Color preservation**: Essential for maintaining natural skin tones in portraits

- **Resolution**: Higher resolutions (1024px) produce better quality but require more computation
- **Batch processing**: Efficiently handles multiple image combinations

8. Conclusion

The pytorch-AdaIN implementation provides an effective and efficient solution for neural style transfer. The Adaptive Instance Normalization technique enables real-time style transfer while maintaining high visual quality. The codebase is well-structured, easy to use, and supports various use cases from single image processing to batch operations and video style transfer.

The model demonstrates the power of feature statistics alignment in transferring artistic styles, making it a valuable tool for both research and practical applications in computational photography and digital art.

9. References

1. Huang, X., & Belongie, S. (2017). "Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization." In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.
2. Gatys, L. A., Ecker, A. S., & Bethge, M. (2016). "Image style transfer using convolutional neural networks." In *Proceedings of the IEEE conference on computer vision and pattern recognition*.
3. Original implementation: https://github.com/xunhuang1995/AdaIN-style
4. PyTorch implementation: https://github.com/naoto0804/pytorch-AdaIN