AInimotion

Speaker Notes & Deep-Dive Study Guide

Everything You Need to Know to Present Confidently

Travis Whitney

February 2026

Contents

1 The Big Picture: What Is AInimotion?

Concept Explained

AInimotion is a **video frame interpolation (VFI)** model purpose-built for **anime**. Given two consecutive frames from an anime video, it generates the frame that should go in between them. This lets you increase the frame rate of anime from 24fps to 60fps or higher, making motion smoother.

The name "AInimotion" = AI + Animation + Motion.

1.1 Why Frame Interpolation Matters

Most anime is drawn at 24 frames per second (fps). Many animators even draw "on twos" — meaning only 12 unique drawings per second. When you watch this on a 60Hz monitor, the motion looks choppy. Frame interpolation fills in the gaps.

Real-world VFI tools like DAIN, RIFE, and SVP already exist and work well on live-action video. But anime is fundamentally different from real video:

- **Flat color regions:** Anime has large areas of uniform color (a character's skin, a blue sky). Optical flow algorithms need texture gradients to track motion — flat regions give them nothing to work with.
- **Non-linear motion:** In live action, a person's arm moves smoothly through space. In anime, a character can "pop" from one pose to another instantly. The in-between frame doesn't exist on a linear path.
- **Ink lines:** Anime's visual identity depends on crisp, thin lines. Standard VFI methods blur these.
- **Scene cuts:** Anime frequently has hard cuts between shots. If you try to interpolate across a scene cut, you get ghostly overlay artifacts.
- **Layer separation:** Anime is literally drawn in layers — static backgrounds that pan behind separately-animated characters. Existing VFI treats the whole frame as one unit.

What to Say

"Anime is uniquely challenging for frame interpolation because it's *drawn*, not filmed. There are no texture gradients for optical flow. Characters jump between poses non-linearly. And most importantly, anime is produced in layers — backgrounds pan rigidly while characters move independently. My model, AInimotion, is the first VFI architecture that explicitly builds this layer separation into the network itself."

2 Slide-by-Slide Speaker Notes

2.1 Slide 1: Title Slide

What to Say

"Hi, I'm Travis Whitney, and today I'm presenting AInimotion — a layer-separated deformable interpolation model designed specifically for anime video frame interpolation."

Pro Tip

Keep this brief — 10–15 seconds max. The title slide just sets the stage.

2.2 Slide 2: Video Frame Interpolation for Anime

This slide introduces the **problem** and why anime is hard.

Concept Explained

Video Frame Interpolation (VFI) is the task of generating intermediate frames between two input frames. Given frames I_1 and I_3, we predict $\hat{I}_2$.
The diagram on the slide shows: Known frame → Generated frame ← Known frame. Arrows point inward because both frames contribute information to the prediction.

What to Say

"The goal is simple: given two known frames from an anime, generate the frame in between. This would let us upscale 24fps anime to 60fps. But anime is uniquely challenging for five reasons."
[Walk through the 5 challenges on the right side of the slide.]
"Why is this a deep learning problem? Because the data has rich structure that deep networks can exploit. Frames have multi-scale spatial structure — edges, textures, objects at different scales — which convolutional networks are designed to capture. There's temporal structure in the motion patterns between consecutive frames. And anime specifically has layer structure: rigid backgrounds vs. deformable characters moving independently. My model encodes all three of these structural priors directly into the architecture. Simple interpolation methods like averaging or linear blending fail catastrophically because motion is non-linear."

2.3 Slide 3: Prior Work & My Contribution

This slide positions your work against existing methods.

Concept Explained

The key prior methods to understand:

1. **DAIN (2019):** Uses depth maps (from a pre-trained depth estimator) to reason about occlusion. If Object A is in front of Object B, DAIN knows to sample from A when synthesizing the in-between frame. *Problem:* Depth estimation models are trained on real photos, not anime art.
2. **AdaCoF (2020):** Instead of computing traditional optical flow (a 2D vector per pixel saying "this pixel moved here"), AdaCoF predicts a $K \times K$ sampling kernel per pixel. Each kernel has offsets (where to look) and weights (how much to use each sample). This is more flexible than a single flow vector. *Problem:* Uses only one motion path — can't distinguish background from foreground motion.
3. **AnimeInterp (2021):** The first anime-specific VFI model. Uses an external segmentation network to separate characters from backgrounds, then handles each differently. *Problem:* Requires a separate, pre-trained segmentation model. If segmentation is wrong, interpolation breaks.
4. **RIFE (2022):** Very fast optical-flow-based VFI. Real-time on consumer GPUs. *Problem:* Optical flow struggles with large, non-linear anime motions. Results are blurry.
5. **LDMVFI (2023):** Uses a latent diffusion model (like Stable Diffusion) to generate interpolated frames. Quality can be excellent. *Problem:* Takes 5–30 seconds per frame vs. AInimotion's ∼50 milliseconds. Not practical.

What to Say

[Point to the table.] "Here's how my approach compares to previous methods."

"DAIN added depth awareness but was designed for real video. AdaCoF introduced deformable kernels, which I actually adopt for my foreground path. AnimeInterp was the first anime-specific model, but it requires an external segmentation network. RIFE is fast but blurs large motion. LDMVFI uses diffusion models and gets good quality but is 100 times slower."

"What makes AInimotion novel are four things: First, I build the dual-path architecture directly into the network — background gets rigid affine motion, foreground gets deformable AdaCoF kernels. Second, this is domain-informed — it mirrors how anime is actually produced in cel layers. Third, I use a *learned* compositor, a soft alpha mask that's predicted by the network, so I don't need any external segmentation. And fourth, I added an edge-aware loss specifically to protect ink lines."

Concept Explained

What does "learned compositor" mean?

AnimeInterp (the prior anime VFI model) required a separate, pre-trained segmentation network to decide which pixels are "foreground" (characters) and which are "background" (scenery). If the segmentation model makes a mistake — say it cuts off part of a character's hair — the interpolation result is permanently broken. That external model was trained on different data and can't be improved by our training.

AInimotion replaces this with a **learned alpha mask predictor** built directly into the model. The compositor module takes the features from both the background path and the foreground path, and predicts a per-pixel value $\alpha \in [0, 1]$:

- $\alpha = 1.0$: "Use the foreground (AdaCoF) result for this pixel"
- $\alpha = 0.0$: "Use the background (affine) result for this pixel"
- $\alpha = 0.5$: "Blend equally" (for ambiguous boundaries)

The key: **nobody tells the model what's foreground or background.** There are no segmentation labels in the training data. Instead, the model discovers this separation on its own because it minimizes reconstruction error. The background affine path *can only* model rigid motion. The foreground AdaCoF path *can* model deformable motion. So the model naturally learns to assign $\alpha \approx 1$ to moving characters (which need deformable handling) and $\alpha \approx 0$ to static/panning backgrounds (which affine handles perfectly). The loss function drives this — whichever assignment produces the best reconstruction wins.

This is what "learned end-to-end" means: the segmentation emerges as a byproduct of optimizing interpolation quality, not from an external model.

2.4 Slide 4: ATD-12K Dataset

Concept Explained

ATD-12K stands for **A**nime **T**riplet **D**ataset with **12,000** samples. Created by the AnimeInterp authors (Siyao et al., CVPR 2021).

Each sample is a **triplet** (I_1, I_2, I_3): three consecutive frames from an anime film. During training, the model sees I_1 and I_3 and tries to predict I_2. The real I_2 is the "ground truth" used to compute the loss.

The dataset is categorized by motion difficulty: small, medium, and large motion. "Large motion" triplets are the hardest — characters may jump several hundred pixels between frames.

How It's Implemented

Training setup details:

- We train on **random** 384×384 **crops**, not full resolution. This saves VRAM and adds data augmentation.
- **Horizontal, vertical, and temporal flips** further augment the data. Temporal flip means swapping $I_1 \leftrightarrow I_3$, which forces the model to learn symmetric interpolation.
- **100K samples per epoch** means the dataloader randomly re-crops 100,000 times per epoch, so the model sees different parts of each image each time.

What to Say

"I train on ATD-12K, the standard anime VFI benchmark. It has 12,000 frame triplets from 30 anime films, categorized by motion difficulty. I use 384-by-384 random crops with horizontal, vertical, and temporal flips for augmentation, drawing 100,000 samples per epoch."

2.5 Slide 5: Architecture Overview

This is the most important slide. Take your time here.

Concept Explained

The LayeredInterpolator is the main model class. The data flows left to right through 6 stages:

1. Input Frames (I_1, I_3): Two RGB images, each $(B, 3, H, W)$, values in $[0, 1]$.

2. Feature Pyramid Network (FPN):

- A **shared encoder** (one set of CNN layers processing both frames with the same weights, so it extracts features the same way for each) processes *both* frames.
- The encoder produces features at 4 scales: $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$ of the input resolution.
- "Feature Pyramid" means we then do a top-down pathway: take the coarsest features (which have the most semantic/global information) and progressively upsample and merge them with finer features. This gives every scale access to both local detail and global context.
- At each scale, we also compute a **correlation volume**. Here's exactly how it works:
  1. **Normalize** both feature maps so each feature vector has unit length (this makes the dot product measure *angle* between vectors — i.e., similarity of what the features "see," regardless of magnitude).
  2. For each pixel (x, y) in frame 1's features, look at a 9×9 search window centered on (x, y) in frame 2's features. That's a displacement range of ± 4 pixels in each direction (in feature space, which is $\frac{1}{2}$ to $\frac{1}{16}$ of full resolution, so ± 4 at $\frac{1}{2}$ covers ± 8 pixels at full resolution).
  3. At each of the $9 \times 9 = 81$ displacement positions, compute the **dot product** between the frame 1 feature vector and the shifted frame 2 feature vector. A high dot product means "these two features look similar," implying the pixel moved to that displacement.
  4. Stack all 81 values into a $(B, 81, H, W)$ tensor: **81 channels**, one per search position.

  Where does 81 come from? The search window is $(2 \times 4 + 1)^2 = 9^2 = 81$. The `max_displacement = 4` parameter controls the search radius. Each output channel encodes "how well does position (x, y) in frame 1 match position $(x + dx, y + dy)$ in frame 2?" for one specific (dx, dy) offset.

  This is fundamentally different from optical flow: instead of predicting one displacement per pixel, we provide the network with a full "matching cost" over all 81 candidate displacements. The downstream networks (background and foreground) then learn to interpret this cost map.

3. Scene Gate:

- Takes the coarsest correlation volume and feeds it through a small neural network (Conv $\rightarrow$ ReLU $\rightarrow$ Pool $\rightarrow$ FC $\rightarrow$ Sigmoid).
- Outputs a confidence score between 0 and 1. If it's below 0.15, we declare a scene cut.
- On scene cut: skip all interpolation and just return frame I_1 directly.
- This prevents the ghostly overlay artifacts you'd get from blending two completely different scenes.

Concept Explained

4. Background Flow (Affine Grid):

- Predicts an 8×8 grid of **affine transformations**.
- Each tile in the grid gets 6 parameters: $[a, b, t_x; c, d, t_y]$ which define rotation, scaling, shearing, and translation.
- Initialized to the identity matrix (no motion).
- The 8×8 grid is upsampled to full resolution via bilinear interpolation, creating a smooth, globally-consistent flow field.
- This models camera motion: pans, zooms, rotations. Exactly what anime backgrounds do.
- Both frames are warped to $t = 0.5$ (the midpoint) and averaged, filling in dis-occluded areas.

What is an affine transformation? An affine transformation is a geometric operation that can do any combination of: translation (shifting), rotation, scaling (zooming), and shearing (slanting). It's defined by a 2×3 matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

These 6 numbers (a, b, t_x, c, d, t_y) control everything:

- **Identity** (no motion): $a = 1, b = 0, t_x = 0, c = 0, d = 1, t_y = 0$ (this is what we initialize to)
- **Pan right by 10px**: $t_x = 10$, everything else identity
- **Zoom in 10%**: $a = 1.1, d = 1.1$, everything else zero/identity
- **Rotate 5°**: $a = \cos(5), b = -\sin(5), c = \sin(5), d = \cos(5)$

Why an 8×8 grid? Rather than one affine transform for the whole image (which would force a single global motion), we divide the image into $8 \times 8 = 64$ tiles. Each tile gets its own affine parameters. This handles *locally varying* camera motion — for example, parallax where the foreground buildings move faster than distant mountains. The 64 tiles are then upsampled to full resolution with bilinear interpolation, creating a smooth flow field with no visible tile boundaries.

Identity initialization: We start all 64 tiles at identity (no motion). This means at the beginning of training, the background path predicts "nothing moved." The network then gradually learns to adjust these parameters to match the actual camera motion in the training data. Starting from identity is crucial — starting from random would produce wild warps on epoch 1.

5. Foreground Flow (AdaCoF Kernels):

- For *every pixel* in the output, predicts a 9×9 kernel with:
  - $9 \times 9 = 81$ sampling offset pairs (dx, dy): where to look in the input
  - 81 weights: how much to use each sample (softmax-normalized)
- Total: $81 \times 2 + 81 = 243$ parameters per pixel.
- This is extremely flexible — each pixel can gather information from 81 different locations in either input frame, at non-integer positions (using bilinear sampling).
- This handles non-linear character motion: squash, stretch, rotation, non-rigid deformation.
- Samples from *both* frame 1 and frame 2, then averages.

Breaking down "81 weights + 162 offsets = 243 params/pixel":

- Think of each output pixel as having its own 9×9 "grabber grid."
- **162 offsets** = 81 positions in the grid × 2 (one for dx, one for dy). These tell the network: "for position #17 in my grid, reach 3.2 pixels left and 1.7 pixels up in the input image." The offsets are continuous (not integer), so we use bilinear interpolation to sample at fractional positions.
- **81 weights** = one weight per grid position, softmax-normalized (so they sum to 1). These tell the network: "position #17 gets 12% of the total influence, position #23 gets 25%",

Concept Explained

6. Compositor (α Blend + Refinement):

- Predicts a soft **alpha mask** $\alpha \in [0,1]$ for every pixel. ("Alpha" comes from compositing terminology — it controls transparency/blending between layers, just like the alpha channel in Photoshop.)
- $\alpha = 1$ means "this pixel is foreground (use AdaCoF result)"
- $\alpha = 0$ means "this pixel is background (use affine grid result)"
- Final blend: output $= \alpha \cdot$ foreground $+ (1 - \alpha) \cdot$ background
- A small **U-Net** refinement network (a CNN shaped like a "U" — it downsamples, processes, then upsamples back to full resolution with skip connections that preserve detail; see Glossary) takes the composite plus both original frames and predicts a **residual correction** (a small pixel-level "fix-up" added to the composite to clean up blending artifacts).
- The compositing is **learned end-to-end** — the entire pipeline is trained as one system, so the model figures out what's foreground and what's background entirely from the data. No segmentation labels needed.

How does the alpha mask predictor work internally?

- **Input:** The finest-scale features from both frames concatenated with the correlation volume. That's $384 + 384 + 81 = 849$ channels (with 96 base channels: the FPN outputs features of $96 \times 4 = 384$ channels at the finest scale).
- **Architecture:** Four convolutional layers: $849 \to 128 \to 64 \to 32 \to 1$, each 3×3 with LeakyReLU activation. The final layer applies a **sigmoid** (squashes output to $[0,1]$).
- **Output:** A single-channel $(B, 1, H, W)$ tensor where each value is between 0 and 1 — the alpha mask.
- **Why it works:** The network learns that areas with high correlation (things that match well between frames) are likely background, while areas with low/complex correlation patterns are foreground (characters with non-rigid motion). It learns this entirely from gradient signals — nobody labeled foreground vs. background.

How does the refinement U-Net work?

- **Input:** 9 channels = the alpha-blended composite (3 RGB) + frame 1 (3 RGB) + frame 2 (3 RGB). Giving it the original frames lets it "see" details that might have been lost during warping.
- **Architecture:** A 3-level U-Net: Encoder $[32 \to 64 \to 128]$ channels with stride-2 convolutions for downsampling, then Decoder $[128 \to 64 \to 32 \to 3]$ with bilinear upsampling + skip connections from the encoder.
- **Residual output:** The U-Net predicts a **residual** — a small correction image that gets *added* to the composite. So the network starts by doing nothing (residual ≈ 0) and learns to fix only what needs fixing: seam artifacts at the alpha boundary, small misalignments, color inconsistencies.
- **Final clamp:** Output is clamped to $[0,1]$ since pixel values must be valid.

Concept Explained

Exact Network Layer Structure (with 96 base channels):
Encoder (shared, processes each frame identically):

- **Level 0** ($\frac{1}{2}$ scale): Conv2d(3 $\rightarrow$ 96, stride=2) $\rightarrow$ BatchNorm $\rightarrow$ LeakyReLU(0.1) $\rightarrow$ ResBlock(96)
- **Level 1** ($\frac{1}{4}$ scale): Conv2d(96 $\rightarrow$ 192, stride=2) $\rightarrow$ BatchNorm $\rightarrow$ LeakyReLU(0.1) $\rightarrow$ ResBlock(192)
- **Level 2** ($\frac{1}{8}$ scale): Conv2d(192 $\rightarrow$ 384, stride=2) $\rightarrow$ BatchNorm $\rightarrow$ LeakyReLU(0.1) $\rightarrow$ ResBlock(384)
- **Level 3** ($\frac{1}{16}$ scale, bottleneck): Conv2d(384 $\rightarrow$ 768, stride=2) $\rightarrow$ BatchNorm $\rightarrow$ LeakyReLU(0.1) $\rightarrow$ ResBlock(768) $\rightarrow$ ResBlock(768)

Each **ResBlock** contains: Conv2d($c \rightarrow c$, 3×3) $\rightarrow$ BatchNorm $\rightarrow$ LeakyReLU $\rightarrow$ Conv2d($c \rightarrow c$, $3{\times}3$) $\rightarrow$ BatchNorm, then **add input back** (skip connection) $\rightarrow$ LeakyReLU.
FPN Top-Down Pathway:

- All levels are projected to $96 \times 4 = 384$ channels via 1×1 lateral convolutions.
- Starting from Level 3: upsample $\rightarrow$ add to lateral of Level 2 $\rightarrow$ smooth with 3×3 conv. Repeat downward.
- Result: 4 feature maps, all 384 channels, at $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$ resolution.

Activation function throughout: LeakyReLU with slope 0.1 for negative values. This means negative inputs are not zeroed out (as in standard ReLU) but instead multiplied by 0.1, preventing "dead neurons."

What to Say

"Let me walk through the architecture. Both input frames first go through a shared Feature Pyramid Network, which extracts features at four scales and computes correlation volumes — essentially a measure of 'where did each pixel move?'
From the FPN, the data splits into two paths. The background path predicts an 8-by-8 grid of affine transformations — each tile captures local camera motion like pans and zooms. This is upsampled to full resolution for a smooth, rigid flow field. The foreground path uses AdaCoF — Adaptive Collaboration of Flows. For every single pixel in the output, it predicts a 7-by-7 kernel with sampling offsets and weights. That's 147 learned parameters per pixel, giving it the flexibility to handle squash, stretch, and non-linear character motion.
A compositor then blends these two layers using a learned soft alpha mask. Where alpha is 1, we use the foreground result; where it's 0, we use the background. A small U-Net does final cleanup.
There's also a Scene Gate that monitors correlation scores. If it detects a scene cut — meaning the two frames are from completely different shots — it bypasses everything and just returns frame 1. This avoids ghosting artifacts."

Pro Tip

Audience calibration: Your professor says the audience knows standard DL concepts (residual layers, U-Nets, BatchNorm, gradient clipping, etc.). Don't explain those. **Do** explain domain-specific concepts they won't know: correlation volumes, AdaCoF kernels, the affine grid, and the layer-separation design.

2.6 Slide 6: Key Components & Loss Functions

Concept Explained

The Loss Function is how the model learns. It measures "how wrong was the prediction?" and the optimizer adjusts weights to minimize it.
The total Generator loss combines four terms:

$$\mathcal{L}_G = \underbrace{2.0 \cdot \mathcal{L}_{L1}}_{\text{Pixel accuracy}} + \underbrace{0.05 \cdot \mathcal{L}_{\text{perc}}}_{\text{Style/texture}} + \underbrace{1.0 \cdot \mathcal{L}_{\text{edge}}}_{\text{Ink lines}} + \underbrace{0.005 \cdot \mathcal{L}_{\text{GAN}}}_{\text{Sharpness (Phase 2)}}$$

What each loss does:

1. **L1 Loss (weight = 2.0):** The L1 loss (also called *Mean Absolute Error*) measures the average absolute pixel difference between the predicted frame $\hat{I}_2$ and the ground truth I_2:

$$\mathcal{L}_{L1} = \frac{1}{N} \sum_{\text{all pixels}} |\hat{I}_2(x,y) - I_2(x,y)|$$

   where N is the total number of pixel values (height $\times$ width $\times$ 3 color channels). Every pixel contributes equally — a 1-unit error on a skin pixel counts the same as a 1-unit error on a background pixel. This is the primary driver of **PSNR** (Peak Signal-to-Noise Ratio), our main quality metric (see Section 3.7). We use L1 instead of L2 (squared error) because L1 produces sharper results — L2 over-penalizes large errors, which causes the network to "play it safe" by predicting blurry averages.

2. **Perceptual Loss (weight = 0.05):** The problem with L1 alone is that it treats every pixel independently. Two images can have low L1 error but look completely different in style or texture. Perceptual loss fixes this by comparing images *through the eyes of a neural network*.
   Here's how it works: We take a **VGG19** network — a deep CNN trained on millions of real photos for image classification (Simonyan & Zisserman, 2014). We chop off the classification head and only keep the internal convolutional layers. When we feed an image through these layers, the early layers detect simple things (edges, colors), and deeper layers detect complex things (textures, shapes, objects).
   We feed both our predicted frame $\hat{I}_2$ and the ground truth I_2 through VGG19, extract feature maps at specific layers (relu1_1, relu2_1, relu3_1, relu4_1, relu5_1), and compare them:

$$\mathcal{L}_{\text{perc}} = \sum_{l \in \text{layers}} w_l \cdot \|\phi_l(\hat{I}_2) - \phi_l(I_2)\|_1$$

   where $\phi_l(\cdot)$ extracts features at layer l, and w_l is the weight for that layer. By comparing features rather than raw pixels, this loss captures *perceptual similarity*: "do these images *look* the same to a neural network?" This preserves artistic texture and style, preventing the washed-out look that L1 alone produces.
   Note: VGG19's weights are *frozen* — they never update during our training. It acts as a fixed "perceptual judge."

Concept Explained

Loss components (continued):

3. **Edge Loss (weight = 1.0):** This is my anime-specific contribution. Ink lines are the visual identity of anime, but they typically occupy only 2–5% of pixels. In a standard L1 loss, the model can afford to blur ink lines because the flat-color majority dominates the loss. Edge loss fixes this by applying **Sobel filters** to detect edges, then dramatically up-weighting those pixels.

   How Sobel filters work: A Sobel filter is a 3×3 convolution kernel that detects intensity gradients (rapid changes in brightness). There are two kernels — one for horizontal edges and one for vertical:

   $$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

   We convolve the ground truth image with both kernels, then combine the results: $|\text{edge}| = \sqrt{G_x^2 + G_y^2}$.
   Pixels where $|\text{edge}|$ is high (ink lines, sharp boundaries) get a weight of 20×, while flat areas keep a weight of 1×. The edge loss is then:

   $$\mathcal{L}_{\text{edge}} = \frac{1}{N} \sum_{\text{pixels}} w_{\text{edge}}(x, y) \cdot |\hat{I}_2(x, y) - I_2(x, y)|$$

   where $w_{\text{edge}} = 1 + 19 \cdot \text{edge\_mask}$. This forces the model to pay 20× more attention to getting ink lines right.

4. **GAN Loss (weight = 0.005):** Only active in Phase 2. A **discriminator** network (a separate CNN) is trained to distinguish real anime frames from generated ones. The generator's GAN loss measures how well it fools the discriminator:

   $$\mathcal{L}_{\text{GAN}} = \text{MSE}(D(\hat{I}_2),\, 1)$$

   The generator wants $D(\hat{I}_2)$ to be close to 1 ("real"). This pushes the output toward the distribution of real anime frames, adding high-frequency crispness and detail that reconstruction losses alone can't capture. The very low weight (0.005) prevents the GAN's adversarial gradients from destabilizing the reconstruction quality built up in Phase 1.

What to Say

"The generator loss has four components. L1 loss with weight 2.0 drives pixel-level accuracy and PSNR. Perceptual loss at weight 0.05 uses VGG19 feature comparison to preserve artistic style. The edge loss is my anime-specific contribution — it uses Sobel filters to detect ink lines and applies a 20-times multiplier on those pixels, forcing the model to preserve them. In Phase 2, a very gentle GAN loss at weight 0.005 adds sharpness without destabilizing training."

2.7 Slide 7: Two-Phase Training Strategy

Concept Explained

Why two phases?
GANs (Generative Adversarial Networks) are notoriously unstable. The generator and discriminator are playing a competitive game: the generator tries to create realistic images, and the discriminator tries to tell real from fake. If you start this game from scratch, neither network knows anything, and the adversarial gradients are essentially random noise — they push the generator in unhelpful directions.

Phase 1 (Epochs 0–34): Reconstruction Only

- Train the generator using only L1 + Perceptual + Edge losses
- No discriminator, no adversarial loss
- Learning rate: 2×10^{-4} (relatively high for fast learning)
- Goal: get the generator to produce *structurally correct* frames (PSNR $\geq$ 28 dB)
- The generator becomes a competent interpolator before ever seeing GAN gradients

D Warmup (500 batches):

- When Phase 2 starts, the discriminator needs to "catch up" to the already-trained generator
- For 500 batches, only the discriminator trains; the generator doesn't see adversarial loss yet
- This prevents the discriminator from giving garbage gradients to the generator

Phase 2 (Epochs 35–49): GAN Fine-Tuning

- Adversarial loss activated with weight 0.005
- Generator learning rate reduced to 5×10^{-5} ($4\times$ lower) to prevent undoing Phase 1's work
- Discriminator trains every other batch (1:2 ratio) so it doesn't overpower the generator
- Goal: add perceptual sharpness and detail

What to Say

"I use a two-phase training strategy. In Phase 1 — the first 35 epochs — I train only the generator with L1, perceptual, and edge losses. No GAN at all. This lets the generator learn stable reconstructions without adversarial interference. The target is PSNR of at least 28 dB.
Then in Phase 2, I introduce the GAN. But I don't just flip it on — there's a 500-batch warmup where only the discriminator trains, so it can catch up to the already-competent generator. After warmup, both train together, but with a reduced learning rate and the discriminator only updating every other batch. This prevents mode collapse and keeps training stable."

2.8 Slide 8: GAN Stabilization & Training Infrastructure

Concept Explained

Why is GAN stability such a big deal?
In anime VFI specifically, the discriminator tends to dominate because anime frames have very consistent visual patterns (flat colors, clean lines). The discriminator quickly learns to detect subtle artifacts, and its gradients overwhelm the generator.

Stabilization techniques I implemented:

1. **D Warmup (500 batches):** Already discussed.
2. **Patch Discriminator:** Instead of classifying the entire 384×384 image as "real" or "fake", the discriminator classifies 70×70 overlapping patches. This focuses on local texture quality ("are these ink lines crisp?") rather than global structure ("is this globally realistic?"). It's more stable and provides better gradient flow.
3. **LSGAN (Least Squares GAN):** Instead of binary cross-entropy (real/fake), we use mean squared error against target values of 0 and 1. Gradients are smoother and training is more stable. From Mao et al., ICCV 2017.
4. **Label Smoothing (0.1):** Instead of training the discriminator with targets of 1.0 (real) and 0.0 (fake), we use 0.9 and 0.1. This prevents the discriminator from becoming overconfident, which would produce near-zero gradients for the generator (the "vanishing gradient" problem).
5. **D Update Ratio (1:2):** The discriminator only updates every other batch. The generator gets twice as many training steps. This prevents D from getting too far ahead of G.
6. **Adaptive lr$_D$:** The discriminator's learning rate automatically adjusts based on its accuracy (d_{acc}). If D is winning too easily (accuracy goes above a threshold), its learning rate decreases. If D is losing, its learning rate increases. This keeps the GAN game balanced.

Training infrastructure:

- **Mixed Precision (FP16):** Uses 16-bit floating point for forward passes to save VRAM. Gradients are computed in 32-bit for numerical stability. PyTorch's `GradScaler` handles the conversion.
- **Gradient Clipping:** If the gradient norm exceeds 1.0, all gradients are scaled down proportionally. Prevents exploding gradients that can crash training.
- **OOM Recovery:** Explained in detail below.
- **Graceful Shutdown:** Pressing Ctrl+C saves a checkpoint before exiting.
- **W&B Logging:** All metrics are logged to Weights & Biases for real-time monitoring.

What is OOM and how does recovery work?
OOM (Out Of Memory) means the GPU ran out of VRAM. During training, the GPU must simultaneously hold:

- The model's weights (~50M parameters × 4 bytes = ~200 MB, doubled for optimizer states)
- The input batch of images (~100–500 MB depending on batch size and crop size)
- All intermediate activations from the forward pass (needed for backpropagation) — this is the biggest consumer, often 10–20+ GB
- The gradients during backpropagation

If any operation pushes total usage above the GPU's 32 GB, PyTorch raises a `RuntimeError` with "CUDA out of memory." Without handling, this crashes the entire training run and you lose all progress since the last checkpoint.

My OOM recovery system works in 4 steps:

1. **Catch the error:** A `try/except` block around the training loop catches any `RuntimeError` containing "out of memory."
2. **Free memory:** Immediately calls `torch.cuda.empty_cache()` to release all cached GPU memory back to the system.

What to Say

"Now let me talk about evaluation — since this is a regression task, not classification, we need appropriate metrics.
The right side of this slide shows our evaluation framework. Our primary metric is PSNR — Peak Signal-to-Noise Ratio. It measures pixel-level accuracy on a logarithmic decibel scale, where every 3 dB improvement roughly halves the mean squared error.
We also track SSIM — Structural Similarity Index — which measures luminance, contrast, and structural patterns. SSIM catches distortions that PSNR might miss, like shifted edges that look wrong but have low pixel error.
And third, visual comparison — side-by-side with ground truth, especially on ink lines and large motion, because quantitative metrics alone can miss perceptual issues.
For infrastructure, we use mixed precision, gradient clipping, OOM recovery, and Weights and Biases for real-time monitoring, all running on an RTX 5090."

2.9 Slide 9: Design Evolution & Training Progress

What to Say

"This table shows how the model evolved through experimentation. I started with 32 base channels and scaled to 96 — three times larger — which significantly improved quality. The kernel size went from 7 to 9 based on our hyperparameter sweep. GAN training originally started from epoch 0, which was unstable, so I moved to the two-phase approach. The L1 weight and perceptual weight were optimized through a 13-experiment sweep. The edge multiplier went from 10x to 20x for sharper ink lines. And I reduced the discriminator learning rate and added the 1:2 update ratio to prevent it from overpowering the generator.
On the right, you can see how my targets compare to existing methods. RIFE achieves around 25 to 27 dB PSNR on ATD-12K — it's fast but blurs large motion. AnimeInterp, the previous anime-specific method, reaches 27 to 29 dB. My target for Phase 1 is at least 28 dB, which would be competitive with AnimeInterp but without requiring an external segmentation network.
The hyperparameter sweep across 13 configurations found that larger crops (384 vs 256), K=9 kernels, and higher perceptual weight all contributed to improved PSNR, with the best configuration reaching 21.80 dB at just 5 epochs."

2.10 Slide 10: Training Iterations & Experimentation

Concept Explained

Full Experiment History — Detailed This slide documents the systematic exploration of approaches, configurations, and architectural decisions throughout the project. Over 30 distinct configurations were tested across 7 major training phases. Below is a detailed breakdown of each phase, what was tried, what went wrong, and what was learned.

Phase 1: 32-Channel Prototype (Jan 11–15)

- Started with the simplest viable model: 32 base channels, a 7×7 AdaCoF kernel, and 256×256 training crops. "Base channels" refers to the width of the first convolutional layer — every subsequent layer in the U-Net encoder doubles this count (32→64→128→256), so the total parameter count grows quadratically with this number.
- Immediately encountered **gradient explosion**: the gradient norm (a single number measuring how "steep" the loss landscape is at the current point) would spike to infinity on certain batches. When this happens, the optimizer takes an infinitely large step, corrupting all learned weights and collapsing the model. In mixed-precision training (AMP), this is even more likely because FP16 has a much smaller dynamic range than FP32 — large intermediate activations overflow to `inf`.
- **Fix**: Implemented **gradient clipping** (`grad_clip=1.0`): after computing gradients but before updating weights, if the total gradient norm exceeds 1.0, all gradients are scaled down proportionally. This caps the maximum step size the optimizer can take, preventing catastrophic weight corruption. AMP requires an extra step: gradients must be "unscaled" (divided by the loss scaler's current scale factor) *before* clipping, because the scaler artificially inflates gradients to maintain precision in FP16.
- Also discovered corrupted triplets in the dataset (truncated JPEG files, missing frames) that caused sporadic crashes during image loading. Added a `try/except` fallback in the data loader that skips bad triplets and returns a random valid one instead.
- Built essential training infrastructure during this phase:
  - **Ctrl+C graceful save**: Catches `KeyboardInterrupt` and saves a checkpoint before exiting, so multi-day training runs can be paused without losing progress.
  - **Batch checkpoints every 5,000 batches**: Saves mid-epoch so that if power is lost or the system crashes, at most ~1 hour of training is lost (on the 5070 Ti).
  - **Auto-resume**: The `--auto-resume` flag automatically finds and loads the latest checkpoint, eliminating the need to manually specify checkpoint paths.
  - **OOM recovery with retry backoff**: If PyTorch throws an "out of memory" error (which can happen for unusually complex batches), the trainer catches the error, frees GPU memory with `torch.cuda.empty_cache()`, saves a checkpoint, waits 10 seconds (increasing by 10s per retry), and then resumes. This allows training to survive occasional VRAM spikes without human intervention.

Phase 2: 48-Channel Scale-Up (Jan 16–19)

- Increased model capacity by scaling base channels from 32 to 48 (a 50% increase). "Capacity" means the model's ability to represent complex patterns — more channels means each layer can learn more distinct features (edge detectors, motion estimators, texture patterns). Trained for 36 epochs on the RTX 5070 Ti (16 GB VRAM), using a batch size of 8.
- Achieved **28.72 dB average test PSNR** when evaluated on 10 full-resolution test triplets. PSNR (Peak Signal-to-Noise Ratio) measures pixel-level accuracy on a logarithmic decibel scale: every +3 dB roughly halves the mean squared error. 28.72 dB is a strong result, competitive with general-purpose VFI methods on anime content.
- **Problem: Discriminator dominance.** In GAN training, the Generator (G) tries to create realistic frames while the Discriminator (D) tries to distinguish generated frames from real ones. Ideally, they improve together — G gets better at fooling D, and D gets

What to Say

“This slide shows the systematic approach I took to get here. This wasn’t a single training run — over the course of this project, I tested more than 30 distinct configurations across 7 major phases.

I started with a small 32-channel prototype to get the basic pipeline working. “Channels” here means the width of the convolutional layers — more channels means the model can learn more features, but also needs more GPU memory. Right away I hit a problem called gradient explosion, where the training gradients would spike to infinity and corrupt all the learned weights. I fixed this by implementing gradient clipping, which caps the maximum step size the optimizer can take. I also built critical infrastructure during this phase: crash recovery, automatic checkpoint saving every 5,000 batches, and OOM error handling.

Then I scaled to 48 channels and got to 28.7 dB on test images after 36 epochs. But I noticed a problem with the GAN training. In a GAN, the generator creates images and the discriminator tries to tell them apart from real images. Ideally they improve together, but my discriminator was winning too easily — hitting 100% accuracy. When that happens, the discriminator’s feedback to the generator becomes useless: it just says ‘everything you make is equally bad’ without telling the generator what specifically to fix.

To understand this, I ran four probe experiments — short targeted runs where I changed one variable at a time. I tried weakening the discriminator, strengthening the GAN signal, and shrinking the model. The key finding was that weakening the discriminator with a lower learning rate helped, but wasn’t enough by itself.

So I ran five more extension experiments across both 32 and 48-channel models, testing different GAN weights and update ratios. An update ratio means the generator gets trained more often than the discriminator, giving it a head start. The counterintuitive insight was that increasing the GAN weight — giving the discriminator more influence — always made things worse. The solution was actually reducing the GAN weight and training the discriminator less frequently.

With those lessons, I built the 64-channel configuration with adaptive GAN balancing. This is a self-tuning system that monitors the discriminator’s accuracy over a window of batches and automatically adjusts its learning rate to keep the accuracy in a healthy range. I also doubled the edge penalization factor, which tells the model to pay 20 times more attention to ink line edges compared to flat background regions.

After upgrading to an RTX 5090 with 32 gigabytes of VRAM, I could scale to 96 channels and run a systematic 13-experiment hyperparameter sweep. I tested learning rates, the balance between pixel-level and perceptual losses, edge handling intensity, crop sizes, and kernel sizes. The kernel size is particularly important because it determines how far the model can ‘reach’ when warping pixels between frames — a 9-by-9 kernel covers a wider search area than 7-by-7. The winning combination was K equals 9, 384 pixel crops, learning rate 3e-4, and boosted perceptual weight. Just changing the crop size from 256 to 384 gave a 0.74 dB improvement, showing that spatial context really matters for motion estimation.

Now the final training is running with all of those optimizations applied, and it’s currently at 22 dB on training crops after 7 epochs, with metrics steadily improving.”

2.11 Slide 11: Summary & Next Steps

What to Say

"To summarize: I built a LayeredInterpolator that separates background and foreground motion — the first VFI architecture to do so. I designed anime-specific edge-aware losses, a stable two-phase training strategy, and robust training infrastructure.
Next steps are completing the full 50-epoch training run with the sweep-optimized configuration, evaluating Phase 2 GAN fine-tuning, and then comparing against RIFE and AnimeInterp baselines on the ATD-12K test set."
The project is open source at github.com/TWhit229/AInimotion."

3 Deep Dive: Key Concepts Explained

3.1 What is Optical Flow?

Optical flow is a 2D vector field that describes how each pixel moves from one frame to the next. If pixel (x, y) in frame 1 moves to $(x + 3, y - 1)$ in frame 2, the flow at that pixel is $(3, -1)$.

Traditional VFI methods estimate optical flow, then **warp** (rearrange) the input frames according to the flow to create the intermediate frame. The problem is that computing accurate flow for anime is extremely hard due to flat color regions.

AInimotion uses flow differently: the **background path** uses coarse affine flow (6 parameters per tile), and the **foreground path** uses deformable kernels (which implicitly learn flow through their offsets).

3.2 What is a Feature Pyramid Network (FPN)?

An FPN is a standard computer vision architecture (Lin et al., 2017). It addresses the scale problem: small features (fine details like ink lines) are best seen at high resolution, while large features (global motion, scene layout) need low resolution.

Bottom-up pathway: A standard CNN encoder progressively downsamples the input, creating feature maps at $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, and $\frac{1}{16}$ of the input size. Each level has more channels (more "knowledge") but less spatial detail.

Top-down pathway: We take the coarsest features (richest semantically), upsample them, and add them to the finer features via lateral connections (1×1 convolutions that align channel dimensions). This gives every level both fine detail AND semantic understanding.

In AInimotion: the encoder uses residual blocks (ResBlocks) with LeakyReLU and BatchNorm. The FPN outputs are all unified to $c \times 4$ channels (where c is the base channel count, default 96). A single shared encoder processes both frames identically.

3.3 What is a Correlation Volume?

A correlation volume measures "how similar is pixel (x, y) in frame 1 to nearby pixels in frame 2?" For each pixel, we search a local 9×9 window (max displacement = 4 pixels at each scale) and compute the dot product between normalized feature vectors. High correlation = likely the same point in both frames.

The result is an 81-channel tensor ($(2 \times 4 + 1)^2 = 81$). This is a compact representation of local motion that downstream modules can consume.

3.4 What is an Affine Transformation?

An affine transformation maps coordinates linearly:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

By choosing (a, b, c, d, t_x, t_y):

- Identity: $a = 1, b = 0, c = 0, d = 1, t_x = 0, t_y = 0$ (no motion)

- Translation (pan): only t_x, t_y are non-zero
- Rotation: $a = \cos\theta, b = -\sin\theta, c = \sin\theta, d = \cos\theta$
- Scaling (zoom): a and d scale proportionally

In AInimotion, there are $8 \times 8 = 64$ tiles, each with its own 6-parameter affine. This allows locally-varying rigid motion (e.g., parallax scrolling where near background pans faster than far background). The weights are initialized to identity (no motion) so the model starts from a safe baseline.

3.5 What is AdaCoF?

AdaCoF = Adaptive Collaboration of Flows (Lee et al., CVPR 2020). Instead of predicting a single flow vector per pixel, it predicts a **sampling kernel**:

For each output pixel, predict K^2 sampling locations (offsets from the pixel center) and K^2 weights. Then:

$$\text{output}(x, y) = \sum_{i=1}^{K^2} w_i \cdot \text{input}(x + \Delta x_i, y + \Delta y_i)$$

With $K = 9$, that's 81 sampling points per pixel. The model can learn to:

- Gather from a single location (like flow): set one weight to 1, others to 0
- Blend from multiple locations (for motion blur or uncertainty)
- Handle sub-pixel shifts (offsets are real-valued; bilinear interpolation is used)

Why not just use flow? Flow says "this pixel came from exactly one place." But in anime, a pixel might be a blend of multiple source locations (transparency, motion blur, artistic choices). AdaCoF handles all of these naturally.

3.6 What is a PatchGAN Discriminator?

A PatchGAN (Isola et al., 2017) classifies overlapping patches rather than the full image. With 3 downsampling layers of stride-2 4×4 convolutions, each output value has a receptive field of 70×70 pixels.

The discriminator output is a spatial map (e.g., 46×46 for a 384×384 input), where each value says "is this 70×70 region real or fake?" The loss is averaged over all patches.

This encourages local texture fidelity (crisp lines, correct color gradients) without forcing global consistency that might conflict with the generator's reconstruction loss.

3.7 What is PSNR?

PSNR (**Peak Signal-to-Noise Ratio**) is the most widely used metric for measuring image reconstruction quality. It answers: "How much noise (error) did my model introduce compared to the maximum possible signal?"

Step 1 — Compute MSE (Mean Squared Error):

First, we measure the average squared difference between the predicted image $\hat{I}$ and ground truth I, across all pixels and color channels:

$$\text{MSE} = \frac{1}{H \times W \times C} \sum_{x=1}^{H} \sum_{y=1}^{W} \sum_{c=1}^{C} \left(\hat{I}(x,y,c) - I(x,y,c)\right)^2$$

where H = height, W = width, $C = 3$ color channels (RGB). MSE = 0 means a perfect reconstruction. Higher MSE = worse quality.

Step 2 — Convert to PSNR:

MSE is hard to interpret directly (is 0.003 good or bad?). PSNR converts it to a logarithmic **decibel (dB)** scale:

$$\text{PSNR} = 10 \cdot \log_{10}\left(\frac{\text{MAX}^2}{\text{MSE}}\right) = 20 \cdot \log_{10}\left(\frac{\text{MAX}}{\sqrt{\text{MSE}}}\right)$$

where MAX is the maximum possible pixel value. For images normalized to $[0, 1]$, MAX = 1, so:

$$\text{PSNR} = 10 \cdot \log_{10}\left(\frac{1}{\text{MSE}}\right) = -10 \cdot \log_{10}(\text{MSE})$$

Why decibels? The dB scale is logarithmic, which matches human perception. A 3 dB improvement means MSE was cut roughly in half. This makes it easier to compare: going from 25 dB to 28 dB is roughly the same "perceptual jump" as going from 28 dB to 31 dB.

Why does optimizing L1 loss improve PSNR? PSNR is defined from MSE (L2), but in practice, minimizing L1 loss also drives PSNR up because both L1 and L2 are minimized when prediction $\approx$ ground truth. L1 just uses a different penalty shape (linear vs. quadratic), which tends to produce sharper images.

Typical PSNR ranges for VFI:

- < 25 dB: Poor, clearly visible artifacts and blur
- 25–28 dB: Decent, some softness visible on close inspection
- 28–32 dB: Good, hard to tell from ground truth at normal viewing
- 32–35 dB: Very good, near-indistinguishable
- > 35 dB: Excellent, essentially pixel-perfect

The target for Phase 1 is ≥ 28 dB ("good" range). Current progress (epoch 9): 22.1 dB and rising.

Limitations of PSNR: PSNR treats all pixels equally. A tiny shift of an ink line by 1 pixel might have low PSNR impact but look obviously wrong. This is why we supplement PSNR with perceptual loss, edge loss, and SSIM.

3.8 What is SSIM?

SSIM (**Structural Similarity Index Measure**) is a metric that evaluates image quality by mimicking how the *human visual system* perceives similarity. Unlike PSNR (which just compares raw pixel values), SSIM compares three properties that humans are sensitive to:

1. **Luminance (l):** Are the images similarly bright overall? Compares the *mean* pixel intensity.

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}$$

2. **Contrast (c):** Do they have similar dynamic range? Compares the *standard deviation* of pixel intensities.

$$c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}$$

3. **Structure (s):** Are the *patterns* similar? Compares the *correlation* between pixel values (after removing mean and normalizing by std deviation).

$$s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}$$

SSIM combines these three components:

$$\text{SSIM}(x,y) = l(x,y) \cdot c(x,y) \cdot s(x,y)$$

SSIM ranges from -1 to 1, where 1 = identical images. In practice, values above 0.95 indicate very high quality. The constants C_1, C_2, C_3 are small stabilizers to prevent division by zero.

How SSIM is computed in practice: SSIM is calculated over small **sliding windows** (typically 11×11 pixels with Gaussian weighting), then averaged across all windows and channels. This makes it a *local* metric — it's sensitive to structural distortions like shifted edges, which PSNR might miss.

Why we track SSIM but don't optimize it directly: SSIM is harder to differentiate cleanly as a loss function (its gradients can be noisy), and L1 + perceptual loss already push the model toward high SSIM indirectly. We use SSIM as an independent "second opinion" metric to verify that our model isn't just gaming PSNR.

3.9 What is L1 Loss vs. L2 Loss?

Both L1 and L2 losses measure the difference between prediction and ground truth, but they penalize errors differently:

- **L1 Loss (Mean Absolute Error):** $\mathcal{L}_{L1} = \frac{1}{N}\sum|\hat{I} - I|$

  Penalty is *linear* with error magnitude. A 2-unit error gets twice the penalty of a 1-unit error.

- **L2 Loss (Mean Squared Error):** $\mathcal{L}_{L2} = \frac{1}{N}\sum(\hat{I} - I)^2$

  Penalty is *quadratic*. A 2-unit error gets *four times* the penalty of a 1-unit error.

Why does this matter for image quality? When the model is uncertain (e.g., a moving character could be in position A or position B), L2 loss encourages predicting the *average* of A and B to minimize worst-case quadratic penalty. This average is a blurry ghost. L1 loss is more forgiving of moderate errors, so the model is more likely to commit to one position — producing a sharper (if slightly wrong) result.

Rule of thumb: L1 $\rightarrow$ sharper but potentially noisier. L2 $\rightarrow$ smoother but blurrier. For anime VFI, sharpness matters more (especially for ink lines), so we use L1.

3.10 How Do Sobel Filters Work?

A Sobel filter is one of the simplest edge detection methods in computer vision. It works by computing the **image gradient** — how fast pixel intensity changes from one pixel to its neighbors.

The two Sobel kernels:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

G_x detects **vertical edges** (intensity changes left-to-right). G_y detects **horizontal edges** (intensity changes top-to-bottom). They work by computing a *weighted difference* between pixels on opposite sides.

Step by step:

1. **Convert to grayscale:** Average the R, G, B channels (or process each separately).
2. **Convolve with G_x:** Slide the kernel across every pixel. At each position, multiply the 3×3 neighborhood by the kernel values and sum. The result tells you "how much does intensity change horizontally here?"
3. **Convolve with G_y:** Same thing, but for vertical changes.
4. **Compute magnitude:** $|\text{edge}| = \sqrt{G_x^2 + G_y^2}$. High magnitude = strong edge.

In AInimotion: The Sobel kernels are registered as *fixed* (non-trainable) convolution filters. The edge magnitude map is then used to create a per-pixel weight map for the L1 loss: $w(x, y) = 1 + (\text{edge\_weight} - 1) \cdot \text{normalize}(|\text{edge}|)$. Edge pixels get $20\times$ higher loss, forcing the model to preserve ink lines.

3.11 How Does VGG Perceptual Loss Work?

VGG19 is a 19-layer convolutional neural network trained in 2014 to classify images into 1,000 categories (dog, car, flower, etc.). Though we don't use it for classification, its *internal layers* have learned incredibly useful visual features:

- **Layer relu1_1 (early):** Detects low-level features — edges, simple color gradients. Very localized.
- **Layer relu2_1:** Detects textures, corner patterns, and simple shapes.
- **Layer relu3_1 (mid):** Detects complex textures, repeated patterns, surface materials.
- **Layer relu4_1:** Detects object parts — eyes, limbs, windows.
- **Layer relu5_1 (deep):** Detects high-level semantic content — faces, objects, scene layout.

How perceptual loss uses VGG:

1. Feed the predicted image $\hat{I}$ through VGG19 (frozen weights, inference only).
2. Feed the ground truth I through the same VGG19.
3. At each chosen layer, extract the feature maps (3D tensors of shape channels $\times$ height $\times$ width).

4. Compute the L1 distance between corresponding feature maps.
5. Weight each layer's contribution and sum.

Why this works: If the predicted image has the same VGG features as the ground truth, it means the images "look the same" to a network that understands visual semantics. This is much more meaningful than pixel-level comparison. For example, a predicted image shifted by 1 pixel would have terrible PSNR but nearly identical VGG features — perceptual loss correctly reflects that it still "looks right."

The VGG normalization trick: VGG19 was trained on ImageNet images normalized with specific mean and std values. Before feeding our images to VGG, we apply the same normalization (mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225]) so the features are meaningful.

3.12 What are BatchNorm and LeakyReLU?

These are standard building blocks used throughout the network:

BatchNorm (Batch Normalization): After each convolution layer, normalize the outputs across the batch so they have mean ≈ 0 and std ≈ 1. This prevents "internal covariate shift" — the problem where earlier layers' changing outputs make it hard for later layers to learn. BatchNorm makes training faster and more stable.

LeakyReLU (Leaky Rectified Linear Unit): An activation function:

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0.1 \cdot x & \text{if } x \leq 0 \end{cases}$$

Unlike standard ReLU (which outputs exactly 0 for negative inputs), LeakyReLU lets a small gradient (10%) flow through for negative values. This prevents "dead neurons" — neurons that get stuck at 0 and never recover.

3.13 What is Gradient Clipping?

During backpropagation, the optimizer computes gradients (partial derivatives of the loss with respect to each weight). Occasionally, gradients can become extremely large ("exploding gradients"), especially in deep networks or with adversarial training.

Gradient clipping is a safety mechanism: before applying gradients, compute the total **gradient norm** (the overall magnitude of the gradient vector):

$$\|\nabla\| = \sqrt{\sum_{\text{all params}} (\nabla w)^2}$$

If $\|\nabla\| >$ max_norm (we use 1.0), scale all gradients down proportionally:

$$\nabla w \leftarrow \nabla w \cdot \frac{\text{max\_norm}}{\|\nabla\|}$$

This preserves the gradient *direction* but limits the step size. It's essential for preventing a single bad batch from crashing training.

3.14 What is Label Smoothing?

In GAN training, we train the discriminator by showing it real images (target = 1.0) and fake images (target = 0.0). If the discriminator becomes *too good* at distinguishing real from fake, two problems occur:

1. Its outputs saturate near 0 or 1 (confident predictions)
2. Gradients near saturation are near-zero (**vanishing gradient problem**)
3. The generator stops receiving useful learning signal

Label smoothing prevents this by softening the targets:

- Real targets: 1.0 → 0.9 (instead of "definitely real," say "probably real")
- Fake targets: 0.0 → 0.1 (instead of "definitely fake," say "probably fake")

This keeps the discriminator from becoming overconfident, ensuring gradients remain informative throughout training.

3.15 What is Mixed Precision Training?

Modern GPUs have specialized hardware (Tensor Cores) for 16-bit floating point (FP16) math. FP16 uses half the memory and computes ∼2× faster than FP32.

Problem: FP16 has limited range (6.5×10^4 max, 6×10^{-8} min positive value) and only 3 decimal digits of precision. Small gradient values can **underflow to zero**, meaning the model stops learning.

Solution: PyTorch's `torch.cuda.amp` module uses "mixed" precision:

- **Forward pass in FP16** — convolutions, matrix multiplies run fast, use half the VRAM.
- **Loss computation in FP32** — prevents accumulated rounding errors in loss.
- **GradScaler** — multiplies the loss by a large number (e.g., 65536) before backpropagation. This "inflates" gradient values so they don't underflow in FP16. After computing gradients, the scaler divides back down before the optimizer step.
- **Weight updates in FP32** — the master copy of all weights is kept in full precision. This is what gets updated by the optimizer.

Result: ∼2× speedup, ∼40% VRAM savings, no quality loss.

4 Glossary of Terms

This section defines foundational terms used throughout the document. If you encounter a term you don't recognize, check here first.

Neural Network Fundamentals

CNN (Convolutional Neural Network): A type of neural network designed for image processing. Instead of connecting every neuron to every pixel (which would be impossibly large), CNNs use small learnable filters (e.g., 3×3) that slide across the image. Each filter detects a specific pattern (edge, color gradient, texture). Stacking many layers of filters lets the network detect progressively more complex features — from edges (layer 1) to textures (layer 3) to objects (layer 10+). Almost every component in AInimotion is a CNN.

Tensor: A multi-dimensional array of numbers. A grayscale image is a 2D tensor (height × width). A color image is a 3D tensor (3 channels × height × width). A batch of images is a 4D tensor (batch × channels × height × width), often written as (B, C, H, W). All neural network computation operates on tensors.

Parameters / Weights: The learnable numbers inside a neural network. A 3×3 convolution filter has 9 parameters. The optimizer adjusts these during training to minimize the loss. When we say "50M parameters," we mean the model has 50 million learnable numbers.

Backpropagation: The algorithm for training neural networks. After computing the loss (how wrong the prediction is), backprop works backward through every layer, computing how much each weight contributed to the error. These partial derivatives are called **gradients**. The optimizer then nudges each weight in the direction that reduces the loss.

Optimizer (Adam): The algorithm that updates weights using gradients. We use **Adam** (Adaptive Moment Estimation), which maintains per-parameter running averages of gradient magnitude and variance. This lets it take bigger steps for parameters with consistent gradients and smaller steps for noisy ones. Adam is the most popular optimizer in deep learning.

Learning Rate (lr): Controls how big each weight update step is. Too high → training oscillates or diverges. Too low → training is painfully slow. Typical values: 10^{-4} to 10^{-3}. We use 2×10^{-4} in Phase 1 and reduce to 5×10^{-5} in Phase 2.

Epoch: One complete pass through the entire training dataset. If the dataset has 10,000 triplets and we draw 100,000 crops per epoch (with random re-cropping), one epoch means the optimizer has seen 100,000 training examples. We train for 50 epochs total (35 Phase 1 + 15 Phase 2).

Batch: A small group of training examples processed together. Our batch size is typically 8–16 images. Processing a batch gives us averaged gradients (more stable than single-image gradients). One epoch consists of many batches.

Overfitting: When a model memorizes the training data instead of learning general patterns. It performs well on training images but poorly on unseen images. Data augmentation (flips, crops) helps prevent this.

Activation Functions & Normalization

Sigmoid: A function that squashes any number into the range $[0, 1]$: $\sigma(x) = \frac{1}{1+e^{-x}}$. Used when we need a probability-like output (Scene Gate confidence, alpha mask values).

Softmax: Converts a vector of raw scores into probabilities that sum to 1. If the AdaCoF kernel predicts raw weights $[2.1, 0.5, 3.0, \ldots]$, softmax normalizes them to $[0.25, 0.05, 0.62, \ldots]$ (sum = 1). This ensures the kernel weights are valid blend proportions.

InstanceNorm (Instance Normalization): Like BatchNorm, but normalizes each image independently (rather than across the batch). Used in the discriminator because the discriminator should judge each image on its own merits, not relative to the batch.

Architecture Components

U-Net: A CNN architecture shaped like the letter "U." It has an encoder (downsampling path) that compresses the image, a bottleneck, and a decoder (upsampling path) that reconstructs the image. Crucially, it has **skip connections** that copy features from the encoder directly to the decoder at matching scales. This preserves fine details that would otherwise be lost during compression. In AInimotion, the refinement network is a small U-Net.

Residual Block (ResBlock): A building block where the input is added back to the output: output $= F(x) + x$. The network only needs to learn the *residual* (the difference from identity), which is easier than learning the full transformation from scratch. This enables training much deeper networks without degradation.

Skip Connection: Any connection that bypasses one or more layers by copying data forward. ResBlocks use additive skip connections ($F(x) + x$). U-Nets use concatenation skip connections (encoder features are concatenated with decoder features). Both help gradients flow backward during training.

Receptive Field: The region of the input image that influences a single output value. In our PatchGAN discriminator, each output value has a 70×70 receptive field, meaning it "sees" a 70×70 patch of the input when making its real/fake decision.

Encoder / Decoder: An encoder compresses input into a compact representation (downsampling, increasing channels). A decoder expands that representation back to the original size (upsampling, decreasing channels). Together they form an autoencoder-like structure.

Training Concepts

Data Augmentation: Artificially increasing dataset diversity by applying random transformations to training images. We use: (1) **Random crops** — cutting 384×384 patches from random positions, (2) **Horizontal flip** — mirroring left/right, (3) **Vertical flip** — mirroring top/bottom, (4) **Temporal flip** — swapping $I_1 \leftrightarrow I_3$ so the model learns interpolation in both directions.

Mode Collapse: A GAN failure mode where the generator learns to produce only one or a few "safe" outputs that fool the discriminator, ignoring the diversity of real data. All generated frames might start looking identical. Our stabilization techniques (low GAN weight, D warmup, label smoothing) prevent this.

Checkpoint: A saved snapshot of the model's weights, optimizer state, and training progress at a particular epoch. If training crashes, we can resume from the latest checkpoint without losing work.

End-to-End Learning: Training the entire pipeline (FPN $\rightarrow$ flow estimation $\rightarrow$ compositing $\rightarrow$ refinement) as one connected system, with gradients flowing from the final loss all the way back to the first layer. The alternative would be training each component separately, which can't optimize the whole system jointly.

Image Processing Terms

Bilinear Interpolation: A method for sampling a pixel value at non-integer coordinates. If AdaCoF wants to sample at position $(3.7, 5.2)$, bilinear interpolation takes a weighted average of the four nearest pixels: $(3, 5)$, $(4, 5)$, $(3, 6)$, $(4, 6)$, with weights proportional to proximity. This gives smooth, continuous sampling.

Warping: Rearranging an image's pixels according to a flow field. If the flow at pixel (x, y) is (dx, dy), warping moves that pixel to $(x + dx, y + dy)$. This is how we "slide" frames to the interpolation midpoint.

Dis-occlusion: When object A moves and reveals part of the background that was hidden behind it. The revealed area has no pixel information in either input frame. Background stitching (averaging warped frames from both directions) helps fill these holes.

Alpha Blending: Combining two image layers using a per-pixel weight: output $= \alpha \cdot$ layer_A $+ (1 - \alpha) \cdot$ layer_B. Used in our compositor to blend foreground and background.

Infrastructure & Tools

VRAM (Video RAM): The GPU's dedicated memory. Training requires storing the model weights, input data, intermediate activations (for backprop), and gradients — all simultaneously in VRAM. Our model uses ∼12–14 GB during training. Mixed precision reduces this by ∼40%.

CUDA: NVIDIA's platform for running computations on GPUs. PyTorch uses CUDA to accelerate tensor operations. "CUDA out of memory" means VRAM is full.

Weights & Biases (W&B): A cloud platform for experiment tracking. During training, we log metrics (loss, PSNR, learning rate, gradient norms) every batch. W&B provides real-time dashboards, making it easy to spot problems like diverging loss or unstable gradients.

PyTorch: The deep learning framework we use. It provides tensor operations, automatic differentiation (computing gradients), GPU acceleration, and pre-built neural network layers. It's the most popular framework in research.

5 Code Structure Quick Reference

File	What It Contains
`models/interp/layered_interp.py`	Main `LayeredInterpolator` class
`models/interp/feature_extractor.py`	FPN: Encoder + top-down + correlation
`models/interp/background_flow.py`	8×8 affine grid predictor
`models/interp/foreground_flow.py`	AdaCoF kernel predictor + sampler
`models/interp/scene_gate.py`	Scene cut detector
`models/interp/compositor.py`	Alpha mask + refinement U-Net
`training/losses.py`	L1, Perceptual, Edge-weighted losses
`training/discriminator.py`	PatchGAN + LSGAN loss
`training/train.py`	Training loop + all infrastructure
`data/dataset.py`	ATD-12K dataloader

6 Potential Q&A

1. **"How does the model know what's foreground vs. background?"**

   It learns this entirely from data via the alpha mask predictor. The network discovers that rigid motion regions (backgrounds) should use the affine path, and deformable regions (characters) should use AdaCoF. No segmentation labels are needed.

2. **"Why AdaCoF instead of just optical flow?"**

   Optical flow assumes each output pixel comes from exactly one source location. In anime, that assumption often fails — pixels can come from multiple sources due to transparency, motion blur, or non-linear pose changes. AdaCoF's kernel formulation naturally handles all of these.

3. **"Why not use a diffusion model?"**

   Diffusion models like LDMVFI produce excellent quality but take 5–30 seconds per frame. AInimotion runs in ~50ms. For practical video upscaling (thousands of frames), speed matters enormously.

4. **"What's the model size?"**

   With 96 base channels: approximately 50–60M parameters. The FPN and AdaCoF networks dominate the parameter count.

5. **"Why not just use RIFE?"**

   RIFE is a general-purpose VFI model. It works reasonably on anime for small motion but significantly blurs large motions and fails to preserve ink lines. AInimotion's dual-path architecture and edge loss specifically address these weaknesses.

6. **"What metric do you optimize for?"**

   Primarily **PSNR** (Peak Signal-to-Noise Ratio), which measures average pixel-level accuracy in decibels (see Section 3.7). The L1 reconstruction loss directly drives PSNR higher. This is supplemented by:

- **VGG perceptual quality** (Section 3.11): Compares deep neural network feature activations between prediction and ground truth, ensuring the output preserves artistic style and texture.
- **Edge preservation via Sobel loss** (Section 3.10): Uses Sobel gradient filters to detect ink lines and applies 20× higher L1 penalty on those pixels, protecting anime's visual identity.
- **GAN loss** (Phase 2 only): An adversarial discriminator pushes output toward the distribution of real anime frames, adding high-frequency crispness.

SSIM (Structural Similarity Index, Section 3.8) is also tracked as an independent metric but not directly optimized. SSIM evaluates luminance, contrast, and structural patterns — it catches quality issues that PSNR might miss, like shifted edges or lost textures.