# A Walkthrough of V-JEPA

DEEP-LEARNING

An interactive, visual walkthrough of the JEPA family of self-supervised architectures — building intuition through I-JEPA's masked image prediction before extending it to V-JEPA's video tubelets and rotary position embeddings.

PUBLISHED
September 28, 2026

20.6 min read

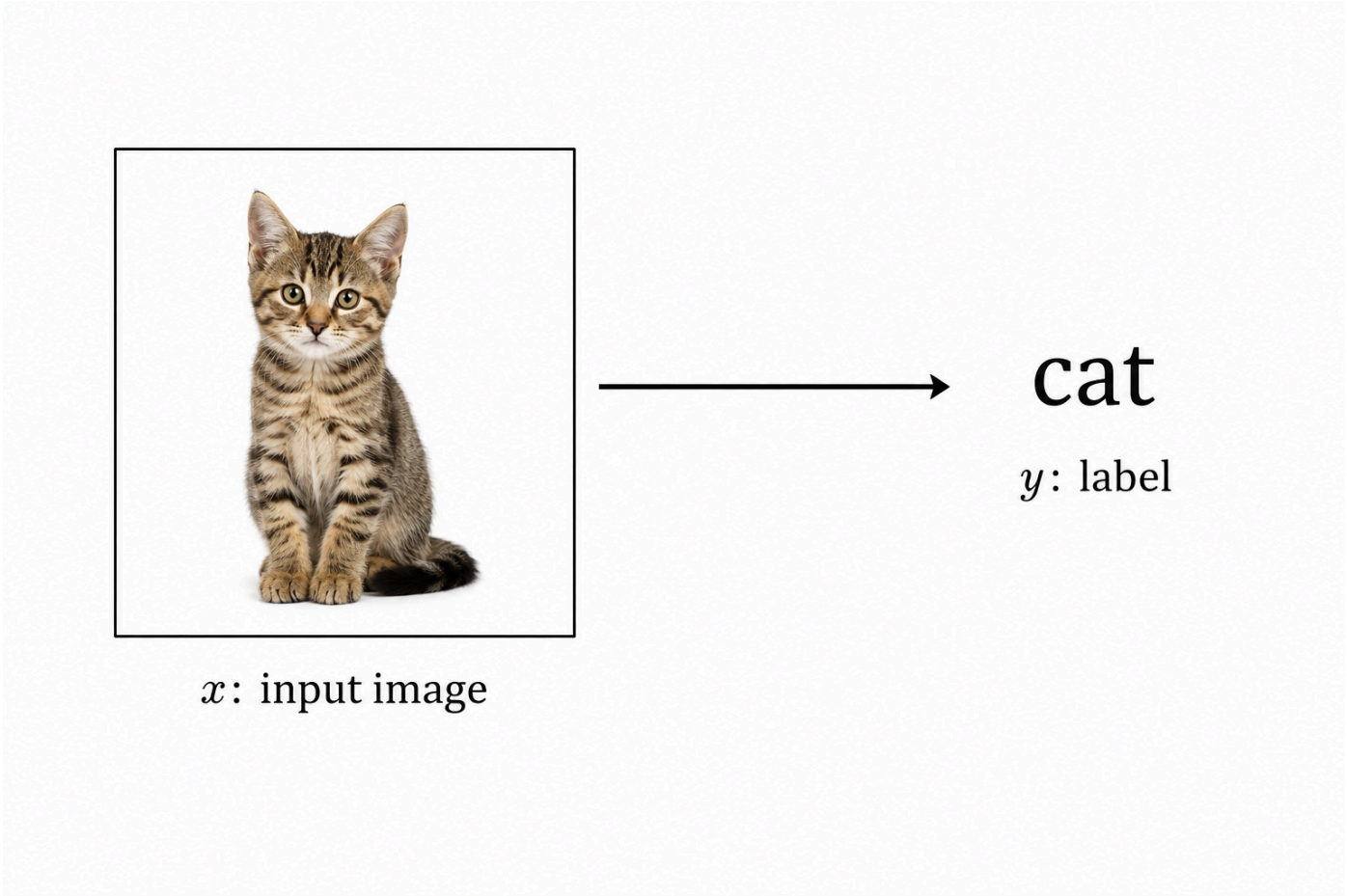
# How to Read this Blog

This blog contains many sections, and is fairly long. The main sections are an overview of SSL, I-JEPA and V-JEPA walk through. Self Supervised Learning is an interesting training paradigm and has differences with respect to supervised learning that are worth pointing out. Once we build a fair intuition for SSL, we introduce the I-JEPA architecture, since this is easier to visualize in some sense than V-JEPA and the intuition transfers fairly neatly. These sections (for the most part) are designed to be self contained (though there will be occasional references to previous sections) so if you're familiar with one section or directly want to jump into V-JEPA you should be able to do so (because of my attempt to isolate these sections, certain points may be repeated multiple times). Also, if you're interested, in the appendix there is a section on representational collapse and how JEPA side-steps this.

# A Brief Overview of Self Supervised Learning (SSL)

## Supervised Learning

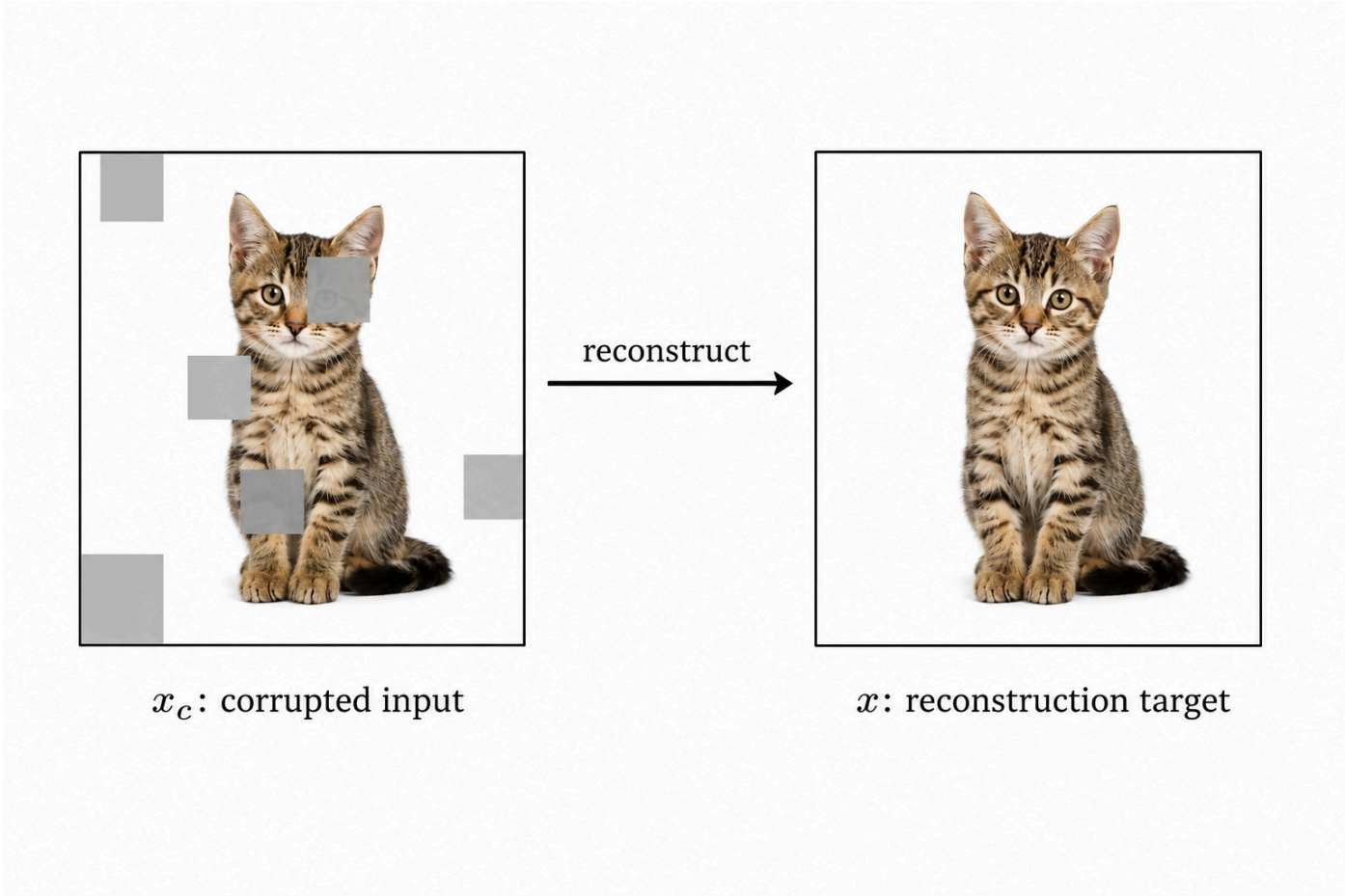
In supervised learning, generally speaking, we have two parts: input data $x$ and a target $y$. The model takes in $x$ and predicts $y$. A good example of supervised learning is image classification, where the model takes in an image $x$ and an annotated label $y$ and tries to predict it.




## Self Supervised Learning

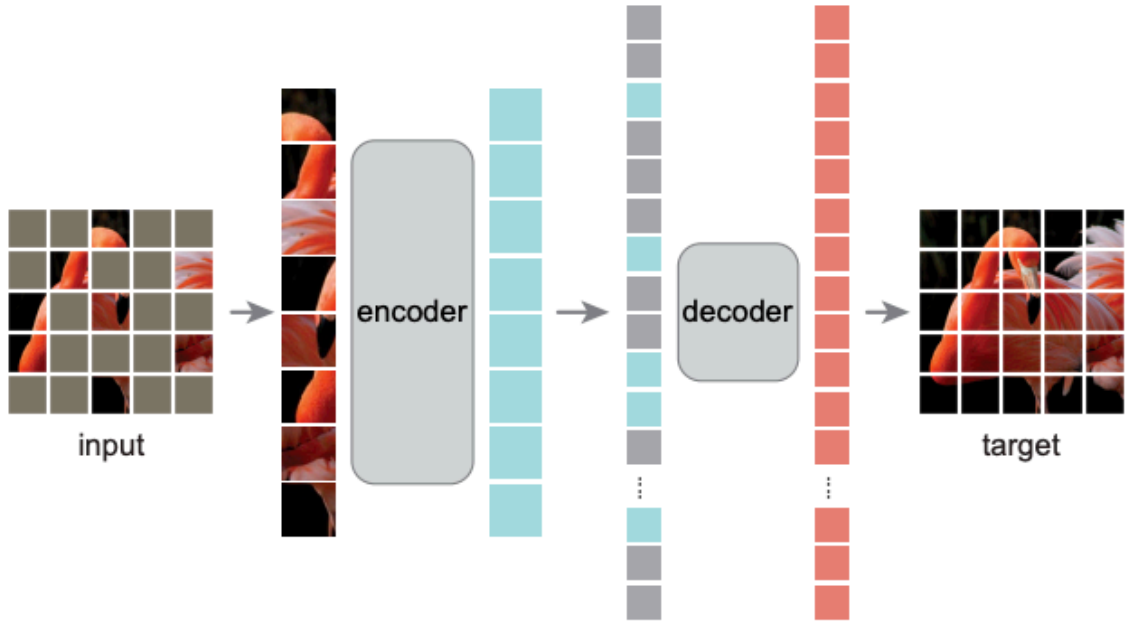
However, JEPA has a different training paradigm called self supervised learning (SSL). In SSL, we don't have any labels, only the input $x$; the learning signal is derived entirely from the data itself, hence the "self". This is beneficial because getting annotated data in the real world is often messy, hard and expensive.

In SSL, we corrupt some part of the input data, so $x$ becomes $x_c$ where the subscript $c$ denotes the corrupted variant. This corrupted version becomes the input during training, and the model is asked to reconstruct the corrupted parts. Typically, the loss function is set up in a way that measures how well the model recovers the original from the corrupted input. Some prominent examples of models/architectures trained under the SSL paradigm are BERT, masked auto-encoders, and contrastive learning.




I should note that corrupting the input is just an example and contrastive learning works a little differently by creating different augmentations/views of the input data and pulling together views of the same sample, and pushing different samples apart. However, for the purposes of this blogpost we'll primarily use masking/corruption as a mental model for SSL because that ties in neatly with what JEPA is trying to do.

Before diving into JEPA, let's take a brief look at Vision Masked Autoencoders (MAE). Here, the model takes in a masked version of the image, and tries to reconstruct the pixels of the masked portion (see image below).[1]



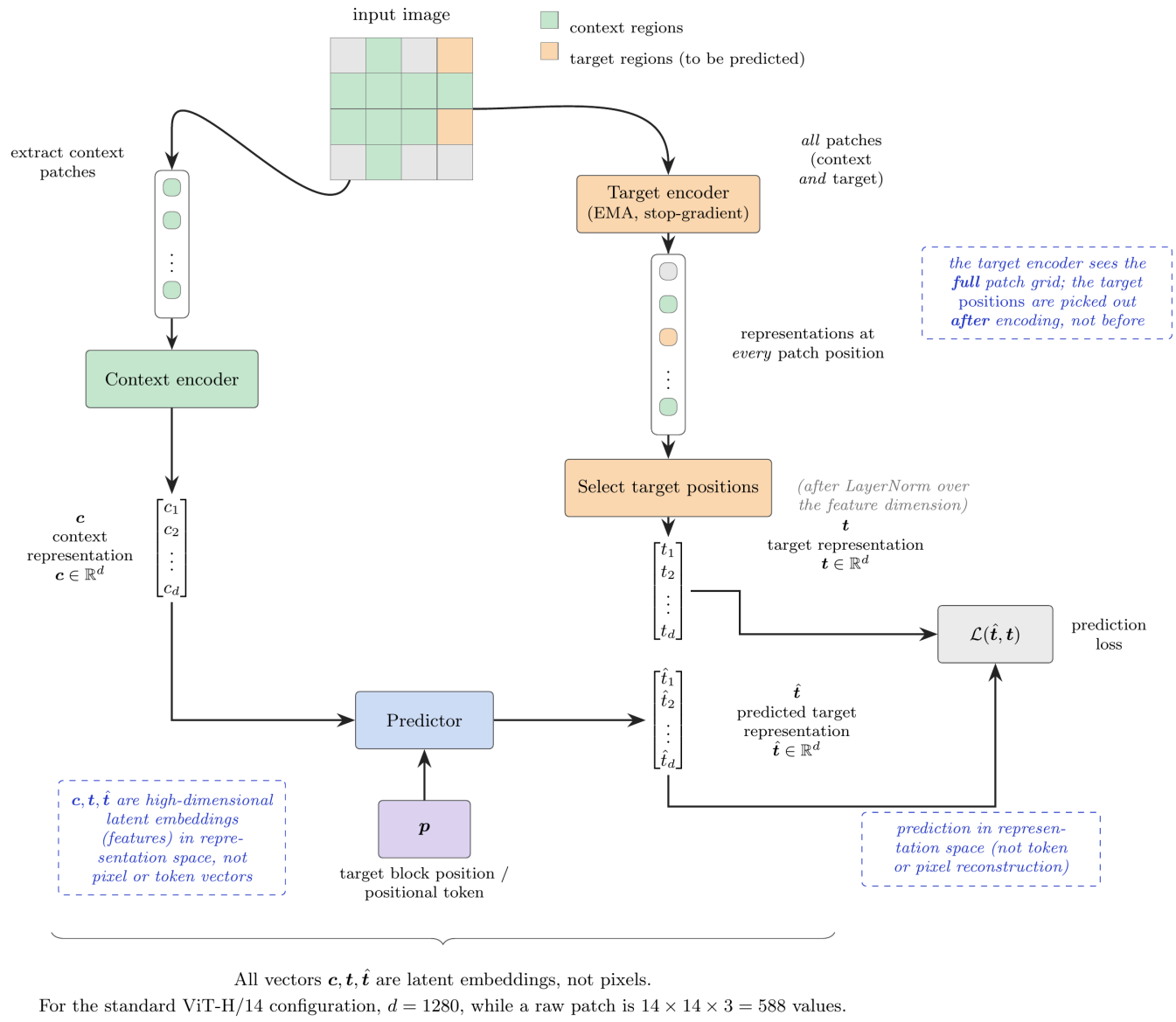

This is exactly the SSL paradigm we've been discussing. Reading the figure left to right: in the input, the grey squares are the masked-out patches, and the remaining visible patches are the context. Only those visible patches go into the encoder, which produces the cyan column; these are latent representations, not pixels. The grey squares that reappear alongside them are the decoder's mask tokens, placeholders standing in for the positions to be filled. The decoder consumes both and produces the pixel reconstruction on the right. However, JEPA's core argument is that predicting at the pixel level is wasteful, because the model has to allocate resources to predict low level details like texture and noise that may not necessarily be semantically relevant. Instead, JEPA proposes that we make our prediction in the latent space. To see how this works in practice, continue to the next section.

# I-JEPA (Image JEPA)

> ⓘ **Note**
>
> While I-JEPA is not directly related to this blog post, I thought it would be useful to include a brief overview of I-JEPA. The motivation behind this is that images are static and are somewhat easier to conceptualize than videos (because of the added dimension). Once we build the intuition for the I-JEPA case, it should hopefully transfer over neatly with some minor modifications.

## Architecture Overview

In the previous section we briefly saw how MAE reconstructs the pixels and that JEPA proposes making predictions in the latent space instead of reconstructing the pixels. Architecturally, I-JEPA would look something like this:




There are three main parts to the JEPA architecture: context encoder, target encoder and predictor network. The context encoder takes the unmasked part of the image and converts it into a context embedding $c$. The target encoder gets the whole image and converts it to a target embedding $t$ (more on this later). The predictor network then takes $(c, p)$ as input where $c$ is the context embedding, and $p$ represents the position of the masked tokens to be predicted and outputs $\hat{t}$. The loss is then calculated with respect to $t, \hat{t}$.

## Training Notes

During training, we follow the architecture shown above, with a few important details concerning how the input image is prepared. Rather than always passing the entire source image to the model, the

preprocessing pipeline first selects a random rectangular crop (note that I-JEPA relies entirely on masking, and does not use multi-view augmentations unlike other SSL models).

Three quantities are relevant here: the output crop size, the crop-area fraction, and the crop aspect ratio. The output size is fixed at $224 \times 224$. The area fraction, denoted by $a$, determines how much of the original image is included before resizing and is sampled from the range $[0.3, 1.0]$. The aspect ratio, denoted by $r$, determines the crop's shape and is defined as the ratio of its width to its height:

$$r = \frac{w}{h}$$

Suppose the original image has height $H$ and width $W$. The desired crop area, measured in pixels, is

$$T = aHW$$

Given $T$ and $r$, the crop dimensions are calculated approximately as

$$\begin{aligned} w &= \text{round}\left(\sqrt{Tr}\right), \\ h &= \text{round}\left(\sqrt{\frac{T}{r}}\right). \end{aligned}$$

Once the crop dimensions have been determined, we randomly select a valid top-left position:

$$\begin{aligned} \text{left} &\in \{0, \dots, W - w\}, \\ \text{top} &\in \{0, \dots, H - h\}. \end{aligned}$$

The resulting rectangular crop is therefore

$$\text{image}\left[\text{top} : \text{top} + h,\ \text{left} : \text{left} + w\right].$$

Use the controls below to see how the area fraction and aspect ratio determine a valid crop, and resample its position within the source image.

*Interactive visualization omitted from the PDF. View it online at interactive/i-jepa/dist/index.html?embed=true&artifact=crop*

Finally, the crop is resized to $224 \times 224$.

*Interactive visualization omitted from the PDF. View it online at interactive/i-jepa/dist/index.html?*

*embed=true&artifact=resize*

In the standard ViT-H/14 configuration, the image is then divided into non-overlapping $14 \times 14$ patches, producing 256 patches arranged as a $16 \times 16$ grid. Each patch is projected into a token representation.

*Interactive visualization omitted from the PDF. View it online at interactive/i-jepa/dist/index.html?embed=true&artifact=patches*

I-JEPA must now determine which token positions will provide context and which will serve as prediction targets. Rather than selecting individual target tokens at random, it selects contiguous rectangular regions, called **target blocks**, on the $16 \times 16$ token grid.

The standard configuration uses four target blocks. I-JEPA samples a target-block area and aspect ratio and uses them to calculate the block's height and width in grid cells. In this implementation, the resulting block size is shared by all four target blocks — and in fact by every image in the batch, since the size is drawn once per training iteration from a seeded generator. Only the locations vary: each block in each image gets its own. Every token inside a target rectangle becomes a prediction target. You might notice that this procedure feels familiar. Sampling the target and context blocks is really the same trick we used earlier when cropping the input image: we choose how much area the region should cover, choose an aspect ratio that fixes its shape, use those to work out a concrete rectangle, and then place that rectangle at a random location. The only thing that has changed is where we apply the idea. Earlier it operated on the raw image in pixel space; here the very same recipe runs on the $16 \times 16$ grid of patch tokens instead, once to lay down the target blocks and once to lay down the context block.

Because the target-block locations are sampled separately, the four blocks may overlap. Consequently, the same token position can appear in more than one target block. Thus, "four targets" refers to four rectangular groups of patch tokens, not four individual patches.

The context region is sampled as a separate, much larger block, covering 85–100% of the grid. Any positions that overlap with the target blocks are removed from the context. The resulting context is therefore not necessarily the complement of the targets, and some tokens may be used by neither branch. Removing the overlapping positions ensures that the context encoder cannot directly observe the tokens it is being asked to predict.

*Interactive visualization omitted from the PDF. View it online at interactive/i-jepa/dist/index.html?embed=true&artifact=masks*

Before the transformer layers process the tokens, fixed two-dimensional sine-cosine positional embeddings are added. These embeddings identify where each token originated on the $16 \times 16$ grid.

The target and context branches then operate differently. The target encoder processes the complete grid of 256 positioned tokens. Only after the full target-encoder forward pass are the representations layer-normalized over the feature dimension, and the representations at the target locations selected. In contrast, the context encoder processes only the positioned context tokens. That layer-norm is a small line of code with outsized importance. It strips the target of any freely chosen scale or offset, which removes one of the easiest routes to a trivial solution.

The predictor receives the encoded context representations together with learned placeholder tokens for the target positions. Each placeholder consists of a shared learned mask token combined with a positional embedding identifying the location it represents. Since one placeholder is provided for every token position in a target block, the complete set of placeholders communicates both the location and shape of the region to be predicted.

Using the context representations and positional placeholders, the predictor produces one representation for each target position. These predictions are then compared with the corresponding representations produced by the target encoder. The comparison is done via a smooth L1 loss, also known as the Huber loss: it behaves like L2 for small errors and like L1 for large ones, which keeps it differentiable at zero while staying robust to outliers.

Now, keen readers might be wondering, what's stopping the model from assigning each masked patch, the same input? Essentially, you're asking the model to create a representation in a high-dimensional space, and then predict that representation (in a way it's like asking a student to write their own questions for an exam and then answer them). If we trained the entire model using backpropagation, this is exactly what would happen. The target encoder would map all the patches to the same point in the embedding space, and then the predictor network would just predict that point, obtaining a loss of zero, but learning nothing useful. That is a serious possibility with self-supervised learning and is a problem called representational collapse, which will be discussed in more detail in the appendix to avoid deviating from the main focus of the blog. Interested readers are encouraged to read that section.

## Inference

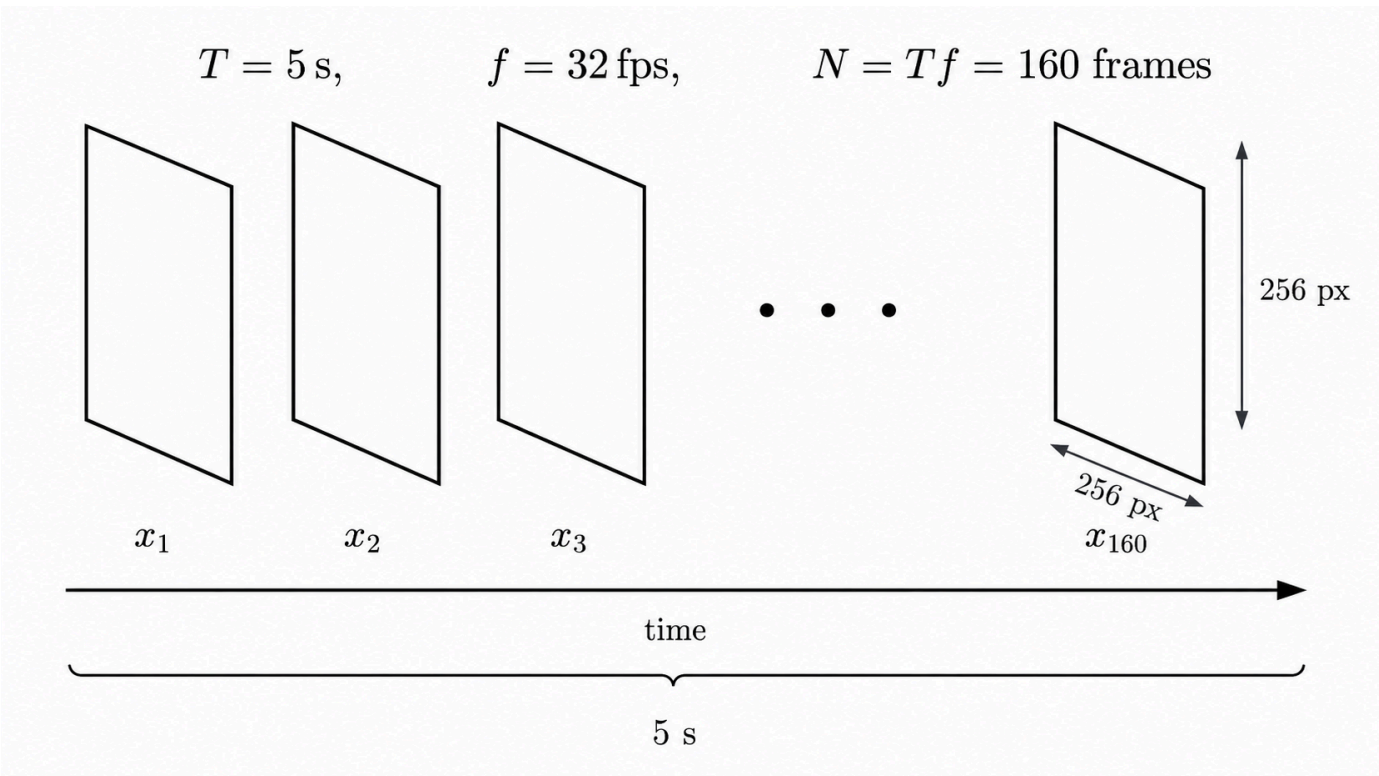
Inference is conceptually a lot simpler. We pass in an image of any size; we resize and center-crop it to 224 by 224. Once we have this resized image, we have 256 patch tokens that get sent into the target encoder. Note this differs from training, which used a *random* resized crop; at inference, we want a single deterministic view, not a sample from a distribution of crops. (The I-JEPA repository ships pretraining code only, so this describes standard practice for evaluating the released checkpoints rather than code in the repo.) We then use these encoded representations for downstream tasks. We no longer need the context encoder and predictor network during inference.

# V-JEPA (Video JEPA)

## Training Notes

> ⓘ **Assumptions**
>
> For ease of explanation we make the following assumptions: a) We ignore batches. Batches don't change the conceptual picture; they just batch together multiple inputs. b) We ignore the color channel for now because it adds another dimensional variable to keep track of.

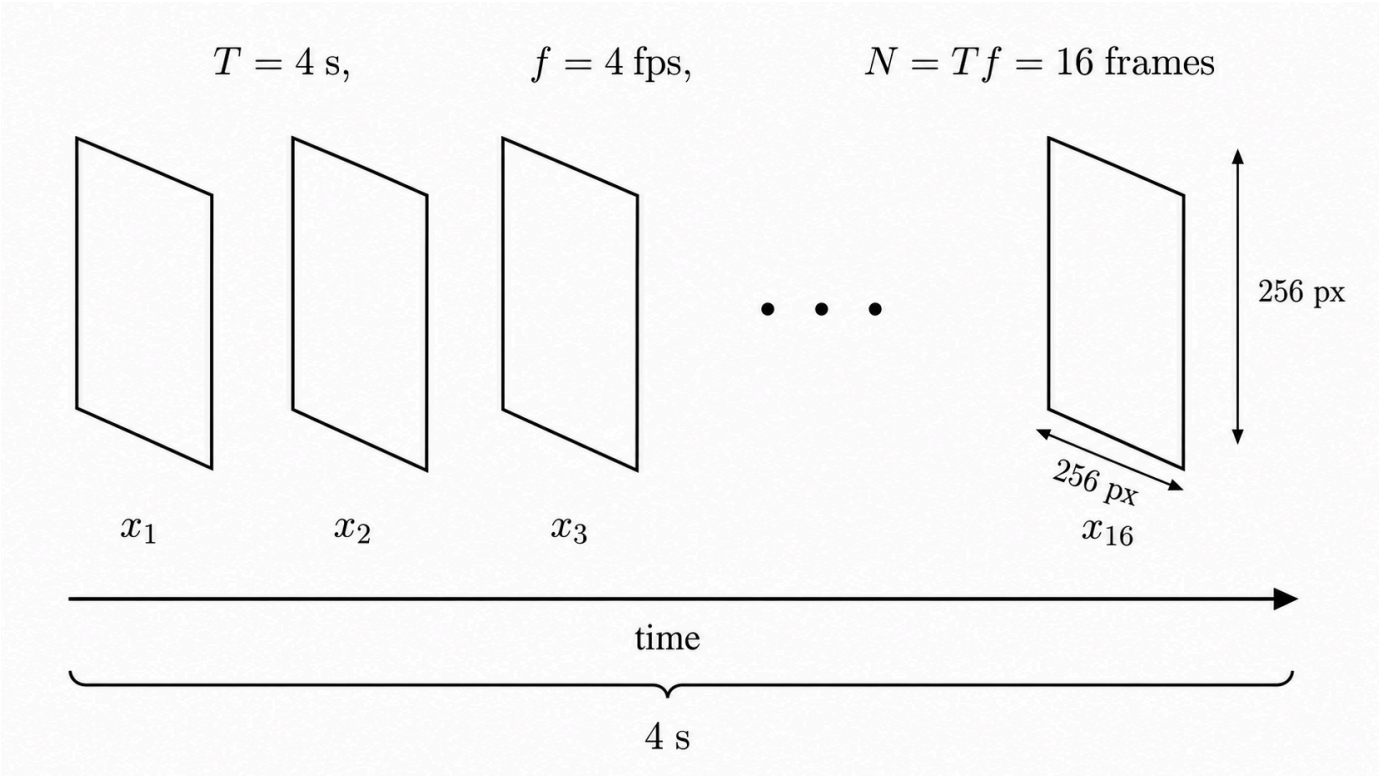
For example's sake, let's say we have a 5 second video, shot at 32fps. That means our video can be split into 160 frames, each of which is 256 by 256. Visually, we can represent it like this:




So our video can be written as $160 \times 256 \times 256$.

Note that we don't pass in our entire video to the model. We first create a sample. To do this, the pipeline first sets the number of frames we want in the sample. The V-JEPA 2 repository sets this to 16 so that's what we'll be working with. It also sets the target fps to 4. So in our example, the sample covers 4 seconds of the original video (16 frames spread across those 4 seconds), not a full-framerate clip, but in general the

length of the video (in seconds) is given by $\text{target frames}/\text{target fps}$ .



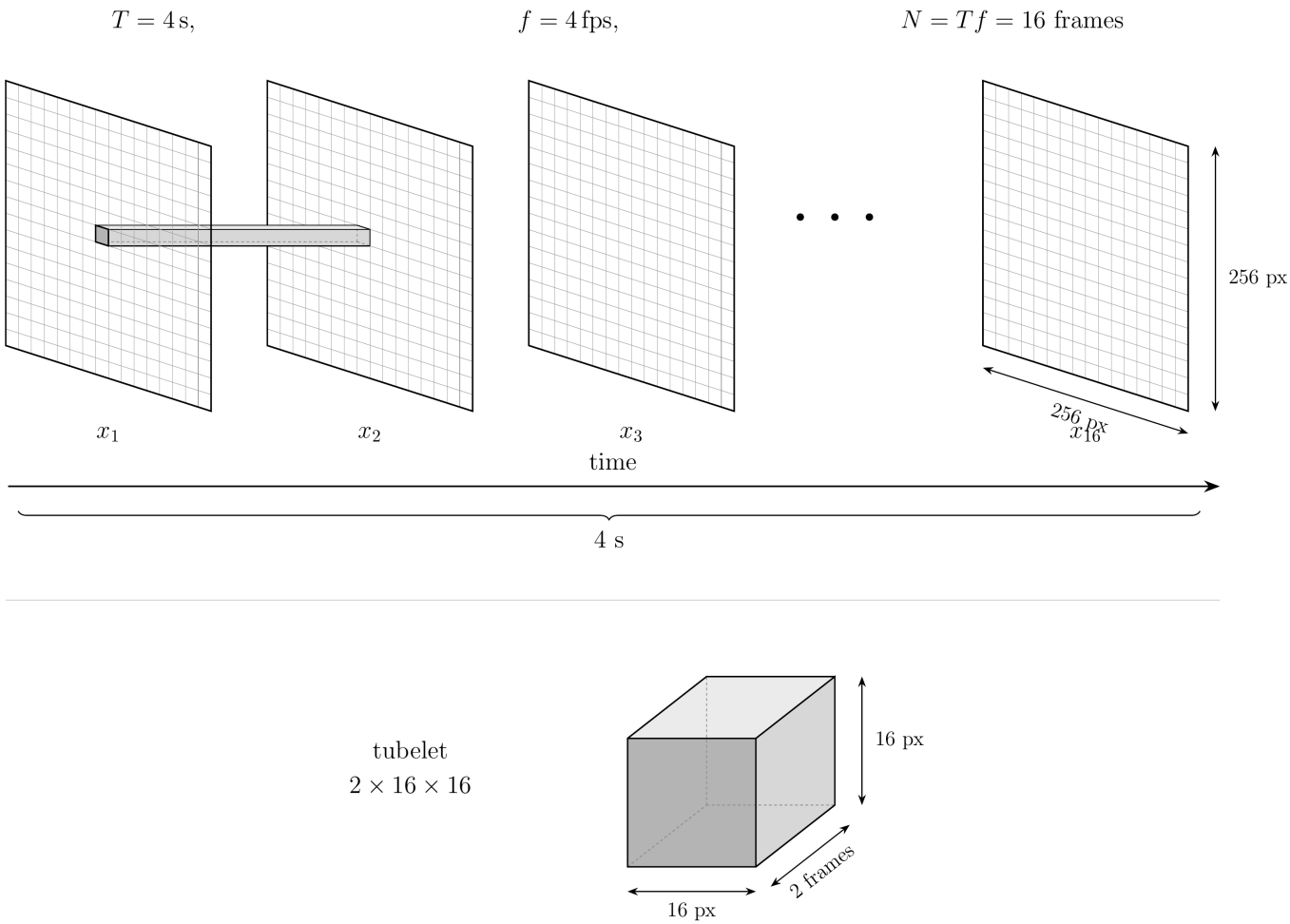

Note that the 4 second window of our sample need not start at 0, so we define a term called the slack which is $\text{source duration} - \text{target duration}$. If the slack is positive, then the start position is picked up randomly from that slack. If the slack is zero, then the window must start at 0. If the slack is negative, no full window exists, so we need to pad. For the purposes of our example, the window can start anywhere between 0 and 1 second into the video, chosen randomly. Of course, longer videos have more potential start positions. Once the start position is chosen, the 16 frames are chosen at regularly spaced intervals. In our example this spacing works out to $\text{source fps}/\text{target fps} = 32/4 = 8$, so we take every 8th source frame within the window (frames 0, 8, 16, and so on).

What we just did was standardizing this across the temporal dimension, and now we normalize the spatial dimensions. The pipeline applies the same crop-then-resize recipe we saw in I-JEPA: a random crop covering 30–100% of the frame with an aspect ratio in $[0.75, 1.35]$, resized to $256 \times 256$, followed by a random horizontal flip. The one important difference from the image case is that a single crop is drawn per sample and applied identically to all 16 frames, so the clip stays temporally coherent — a per-frame crop would introduce motion that isn't in the source video.

Now that we've formed our sample, we need to form our input to the model. The input in the V-JEPA model is what we call a tubelet. A tubelet is a patch of a frame, tracked across two sequential frames in a given sample. Note that these are not necessarily adjacent frames in the input video, but they are adjacent in our sample. Note that each frame can be represented as a grid of 256 patches, each patch being $16 \times 16$. This

patch, tracked across two frames, forms a tubelet. So for our example then we have $8 \times 16 \times 16$ tubelets, and each tubelet is $2 \times 16 \times 16$.

Visually, we can represent a tubelet like this:



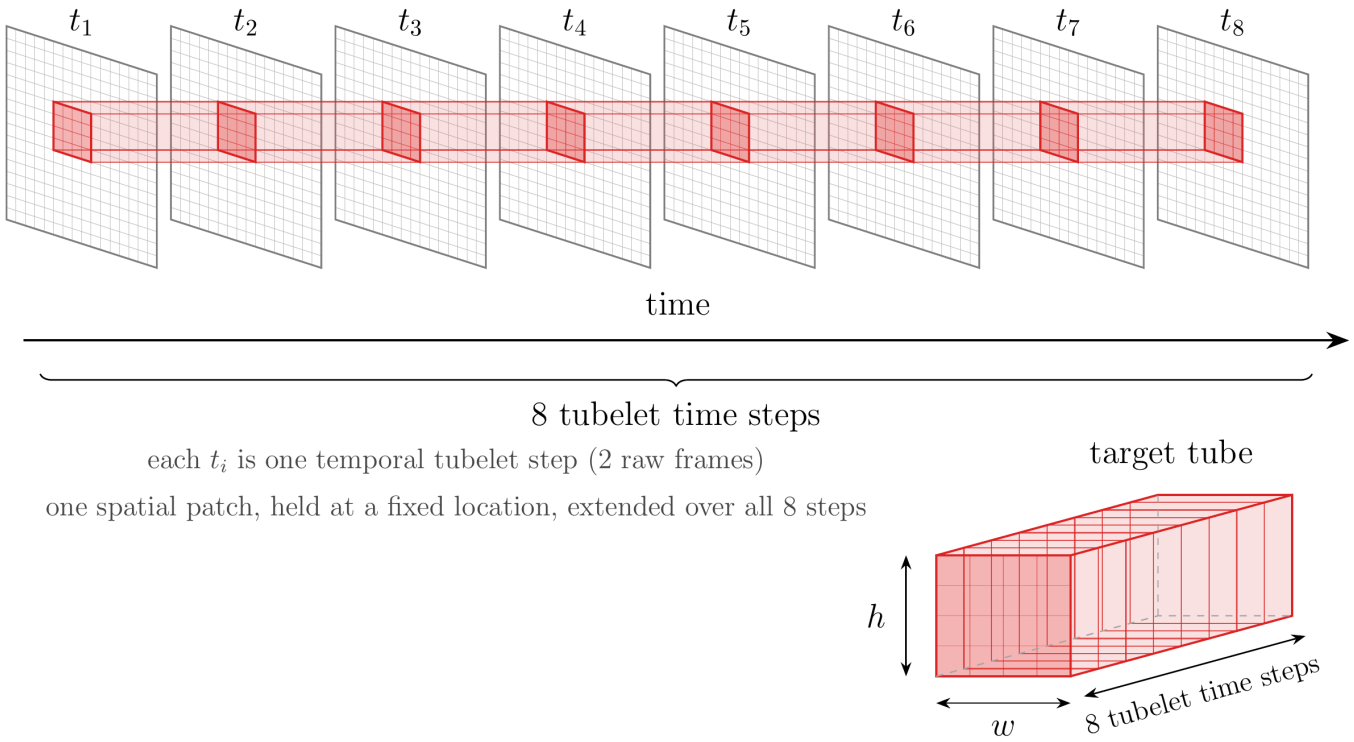

Now, we flatten this tubelet and pass it through a linear projection to create our input vector. As of now, this has no positional representation.

In the original V-JEPA, this is where a fixed sinusoidal positional encoding would get added directly to the token embedding. V-JEPA 2's released models don't do this; instead, they use **RoPE (Rotary Position Embeddings)**. Rather than encoding position once, additively, at the embedding stage, RoPE injects positional information inside every attention layer by rotating the query and key vectors according to each token's (time, height, width) coordinates. We won't get into the mechanics of RoPE here since that's not the focus of this post; the important point for this walkthrough is just *where* position enters the model: not as a vector added to the input, but as an operation applied inside attention, at every layer.

If you're interested in learning more about RoPE, here's a video I found helpful.[2]

Now we have 2048 tokens.

Now that the tokens are formed, we're ready to create our target and context patches. In order to do that, we take random spatial patches and form tubes that span over the temporal frames (the temporal frame depth is something that we configure, but the shipped V-JEPA 2 implementation spans all 8 tubelets). The image here should make this clearer:




To form the target patches, we set a configurable parameter representing the number of **blocks** we form (the shipped V-JEPA implementation sets this to 8). To form each block's spatial extent, we use a strategy similar to I-JEPA, starting from an area and an aspect ratio and combining it with a sampled temporal extent, giving a 3D tube. To avoid confusion, note that the tubelet extends across frames, but the tubes extend across the tubelets. We stamp several such tubes at random positions; their union is the target region, and everything not masked out serves as context (V-JEPA 2 actually applies two such masks per sample: one with 8 small tubes covering roughly 15% of the spatial grid each, and one with 2 large tubes covering roughly 70% each. Everything described here is the small-tube mask; the large-tube mask is omitted as an intentional simplification). Essentially, the context is the complement of the union of the target tubes.

From here, the process is similar to I-JEPA, and the effort we put in earlier finally pays off here. For a quick summary, the target encoder sees all tokens; the context encoder only sees the context tokens; and the predictor network then takes the context encoder output along with positions and predicts the target, which is compared with the output of the target encoder. As in I-JEPA, each target position is supplied to the predictor as a learned placeholder token, but two details differ. V-JEPA 2's predictor holds a separate mask token for each mask configuration — two of them, matching the two masks above — rather than one shared token. And because the model uses rotary position embeddings, the placeholder carries no added

positional embedding; its position enters through the rotation applied inside attention. The comparison is done via the L1 loss. Note this comparison happens in representation space, not pixel space, which is the latent-prediction idea from earlier made concrete. Note that the target encoder isn't a separate network, it's a slowly-moving copy of the context encoder (updated as a running average of its weights rather than by gradients). The mechanics of this update, and how it works together with stop-gradient to guard against representational collapse, are in the appendix.

## Inference

The learnings from I-JEPA also carry forward here during inference. The predictor network is discarded, and the target encoder is used as a feature extractor.

# Sources


1. Masked auto encoders are scalable vision learners.
2. RoPE: Understanding rotary positional embeddings in transformers.
3. Self-supervised learning from images with a joint-embedding predictive architecture.
4. I-JEPA official repository (facebookresearch/ijepa).
5. Revisiting feature prediction for learning visual representations from video.
6. V-JEPA official repository (facebookresearch/jepa).
7. V-JEPA 2: Self-supervised video models enable understanding, prediction and planning.
8. V-JEPA 2 official repository (facebookresearch/vjepa2).


# Appendix

## Representational Collapse

> ⓘ **Notes**
>
> 1. This section isn't necessary to understand the core details of how JEPA works, but it helps explain the training process.
> 2. This should not be treated as a formal proof. We are a bit loose with some definitions, but the core idea should conceptually hold.

Firstly, by representational collapse, we mean the model admits a trivial solution in which the encoder outputs the same representation for all data points, achieving a minimal loss while learning nothing about the input data.

To work with this further, let's use a toy model as a useful abstraction. Let's define our model as follows:

$$t_i = W_1 x_i$$
$$\hat{t}_i = W_2 x_i^{'}$$

where $t_i$ is the output of the target encoder for data point $x_i$ and $\hat{t}_i$ is the predictor network output and $x_i^{'}$ is the corrupted version. Here we're training with the naive version where $W_1, W_2$ are updated using gradient descent on the same loss with no stop gradient. In this toy model, we'll also make one more modification and use an $L_2$ loss function. Note that these modifications are chosen for ease of computations but they don't change the underlying intuition all that much. Also note that here $i$ indexes over a data point; so $t_i, x_i$ are vectors.

We can write our average loss for $N$ data points as:

$$l_2 = (1/N) \sum_{i=1}^{N} (t_i - \hat{t}_i)^2$$
$$l_2 = (1/N) \sum_{i=1}^{N} (W_1 x_i - W_2 x_i^{'})^2$$

Now, we want to take the derivative of the loss function with respect to the parameters. Let's start with $W_1$.

$$\begin{aligned}
\frac{\partial l_2}{\partial W_1} &= (1/N) \sum_{i=1}^{N} \frac{\partial}{\partial W_1} (W_1 x_i - W_2 x_i^{'})^2 \\
&= (1/N) \sum_{i=1}^{N} 2(W_1 x_i - W_2 x_i^{'}) \frac{\partial W_1 x_i}{\partial W_1} \\
&= (1/N) \sum_{i=1}^{N} 2(W_1 x_i - W_2 x_i^{'}) x_i^{\top} \\
&= (2/N) \sum_{i=1}^{N} (W_1 x_i - W_2 x_i^{'}) x_i^{\top}
\end{aligned}$$

Likewise, we can repeat this for $W_2$:

$$
\begin{aligned}
\frac{\partial l_2}{\partial W_2} &= (1/N)\sum_{i=1}^{N}\frac{\partial}{\partial W_2}(W_1x_i - W_2x_i')^2 \\
&= (1/N)\sum_{i=1}^{N}2(W_1x_i - W_2x_i') \times (-\frac{\partial W_2x_i'}{\partial W_2}) \\
&= (1/N)\sum_{i=1}^{N}-2(W_1x_i - W_2x_i')x_i'^{\top} \\
&= (-2/N)\sum_{i=1}^{N}(W_1x_i - W_2x_i')x_i'^{\top}
\end{aligned}
$$

Note, that with $W_1$ held fixed, the loss is minimized where the surface is flat (i.e. gradient is 0, the loss surface is convex), so we set $\frac{\partial l_2}{\partial W_2} = 0$

$$
\begin{aligned}
\frac{\partial l_2}{\partial W_2} &= 0 \\
(-2/N)\sum_{i=1}^{N}(W_1x_i - W_2x_i')x_i'^{\top} &= 0 \\
\sum_{i=1}^{N}(W_1x_i - W_2x_i')x_i'^{\top} &= 0 \\
\sum_{i=1}^{N}(W_1x_i)x_i'^{\top} - \sum_{i=1}^{N}(W_2x_i')x_i'^{\top} &= 0 \\
\sum_{i=1}^{N}(W_1x_i)x_i'^{\top} &= \sum_{i=1}^{N}(W_2x_i')x_i'^{\top} \\
W_1\sum_{i=1}^{N}x_ix_i'^{\top} &= W_2\sum_{i=1}^{N}x_i'x_i'^{\top} \\
W_1\underbrace{\sum_{i=1}^{N}(x_ix_i'^{\top})}_{B} &= W_2\underbrace{\sum_{i=1}^{N}x_i'x_i'^{\top}}_{A} \\
W_1B &= W_2A
\end{aligned}
$$

Now, we can isolate $W_2$, and we get:

$$
W_1BA^{-1} = W_2
$$

Let's plug this back into our gradient equation for $W_1$.

$$
\begin{aligned}
\frac{\partial l_2}{\partial W_1} &= (2/N)\sum_{i=1}^{N}(W_1 x_i - W_2 x_i')x_i^\top \\
&= (2/N)\sum_{i=1}^{N}(W_1 x_i - W_1 BA^{-1}x_i')x_i^\top \\
&= (2/N)\sum_{i=1}^{N}(W_1 x_i x_i^\top - W_1 BA^{-1}x_i' x_i^\top) \\
&= (2/N)W_1\sum_{i=1}^{N}(x_i x_i^\top - BA^{-1}x_i' x_i^\top) \\
&= (2/N)W_1[\sum_{i=1}^{N}(x_i x_i^\top) - BA^{-1}\sum_{i=1}^{N}(x_i' x_i^\top)] \\
&= (2/N)W_1[\underbrace{\sum_{i=1}^{N}(x_i x_i^\top)}_{C} - BA^{-1}\underbrace{\sum_{i=1}^{N}(x_i' x_i^\top)}_{B^\top}] \\
&= (2/N)W_1[C - BA^{-1}B^\top]
\end{aligned}
$$

Now, when we use gradient descent to update $W_1$ we get:

$$
\begin{aligned}
W_1 &\leftarrow W_1 - \eta\frac{\partial l_2}{\partial W_1} \\
&\leftarrow W_1 - \eta(2/N)W_1[C - BA^{-1}B^\top] \\
&\leftarrow W_1(I - \eta(2/N)[C - BA^{-1}B^\top]) \\
&\leftarrow W_1(I - \eta(2/N)\underbrace{[C - BA^{-1}B^\top]}_{S}) \\
&\leftarrow W_1(I - \eta(2/N)S)
\end{aligned}
$$

Let's say we define $I - \eta(2/N)S$ to be $M$, then we get the update rule as

$$W_1 \leftarrow W_1 M$$

and after k updates, we have:

$$W_1^{(k)} = W_1^{(0)} M^k$$

Note that $N, A, C, B$ are defined on the inputs, so they are considered fixed from a learning perspective.

Now, the problem here is that we're multiplying $W_1$ by $M$ repeatedly, so it could tend to zero. To understand this, let's assume for a second that we're dealing with scalars.

$$w_1 \leftarrow w_1(1 - \eta(2/N)s)$$

For example's sake, let's say $\eta(2/N)s = 0.1$ and $w_1 = 1$ then from the equation we derived above we get:

$$w_1^{(k)} = w_1^{(0)}(1 - 0.1)^k$$

Over many $k$ steps, $w_1 \to 0$ which means that it effectively doesn't learn anything related to the data. Or more formally $t_i = w_1 x_i \to 0$ for every input $x_i$ so every data point gets the same representation.

To look at this for matrices, we can take a look at their eigenvalues. To do this, we consider the eigenvectors of $S$ and their corresponding eigenvalues $\lambda$. The eigenvectors split $W_1$ into directions, and give us a way to reason about what the weights learn from different directions of input data. We can then use the scalar argument presented above, only that we swap $s$ with the eigenvalue $\lambda$ for that direction. If all eigenvalues are greater than 0, that leads to full collapse as in the scalar example above. If only some eigenvalues are greater than 0, then the model does not learn anything in that direction.

So to fix this representational collapse issue, we use stop gradient along with exponential moving average (EMA) to provide the weight updates for $W_1$. This means, we no longer compute the gradient for $W_1$, and therefore do not update the weights using gradient descent. In fact, we update the weights using a method called EMA, which updates the weights of the target encoder using the weights of the context encoder. So our new update rule becomes:

$$W_1 \leftarrow \tau W_1 + (1 - \tau) W_c$$

where $W_c$ are the weights of the context encoder (our toy model doesn't separate out the context encoder, so this update describes the real architecture).

In the naive version, it's like asking a student to write their own exam, answer it, and grade it themselves. The easiest way to get a perfect score is to write questions that all have the same answer, and then give that answer every time. The score is perfect, but the student hasn't learned anything. However, in this new setting with EMA, we don't face this problem because the target is no longer written by the student and the only way to score well is to actually learn. Note that collapse is still possible; it's just that the model does not have a strong incentive to go there anymore.

Note that this is not a problem that the MAE faces, since the MAE is asked to predict directly at the pixel level. Other SSL methods are vulnerable to their version of representational collapse.