SC 4001 CE/CZ 4042 Neural Network and Deep Learning

Last update: 15 April 2024

Revision & Applications of GANs

Xingang Pan

潘新钢

https://xingangpan.github.io/

https://twitter.com/XingangP

Revision

- **Week 7** – Convolutional Neural Network (CNN) I
- **Recess Week**
- **Week 8** – Convolutional Neural Network (CNN) II
- **Week 9** – Recurrent Neural Networks (RNN)
- **Week 10** – Attention
- **Week 11** – Autoencoders
- **Week 12** – Generative Adversarial Networks (GAN)
- **Week 13** – Revision and Selected Topics

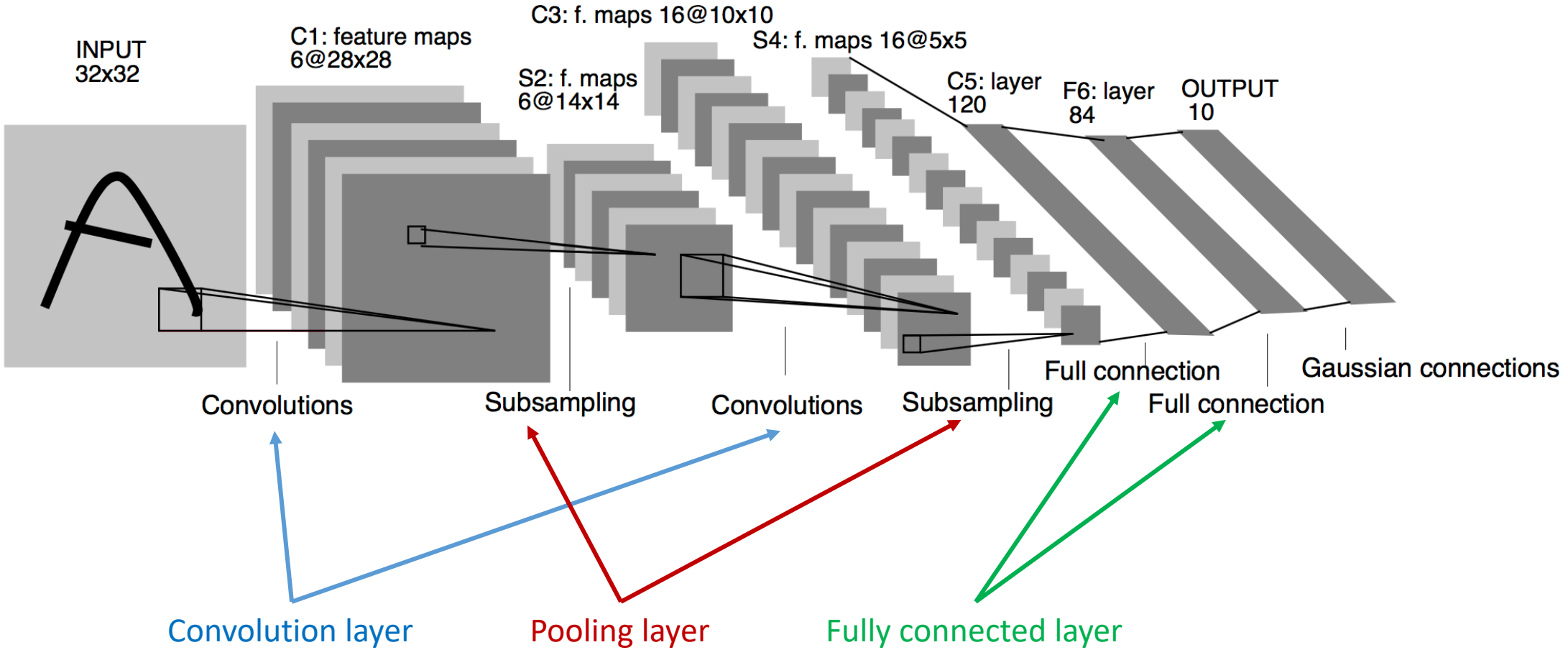
Final Exam

- Time: 9:00 AM – 11:00 AM, 7 May 2024
- Venue: Hall C
- The exam will be open book

Outline – CNN I

- Basic components in CNN
- Training a classifier
- Optimizers

An example of convolutional network: LeNet 5

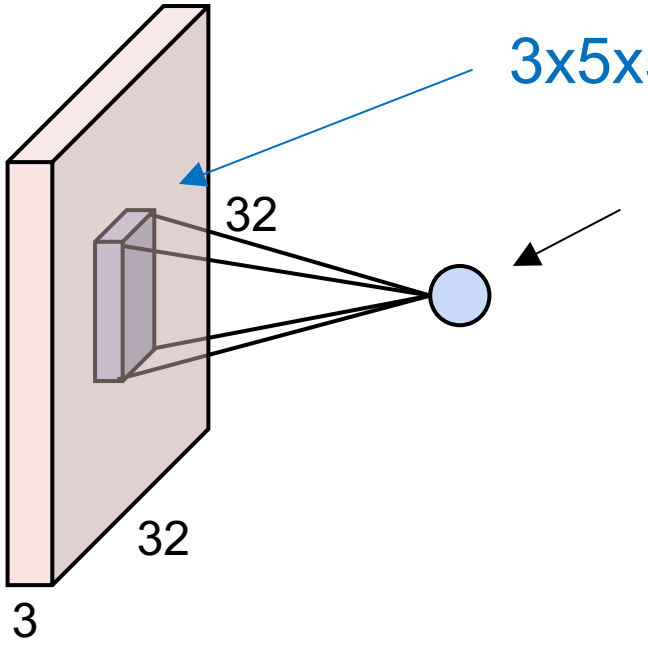


As we go deeper (left to right) the height and width tend to go down and the number of channels increased.

Common layer arrangement: Conv → pool → Conv → pool → fully connected → fully connected → output

Convolution layer

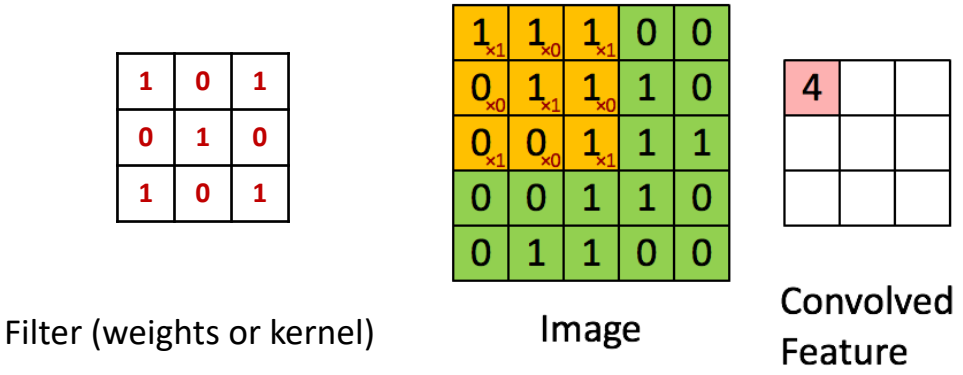
3x32x32 image



1 number:
the result of taking a dot product between the filter and a small 3x5x5 chunk of the image

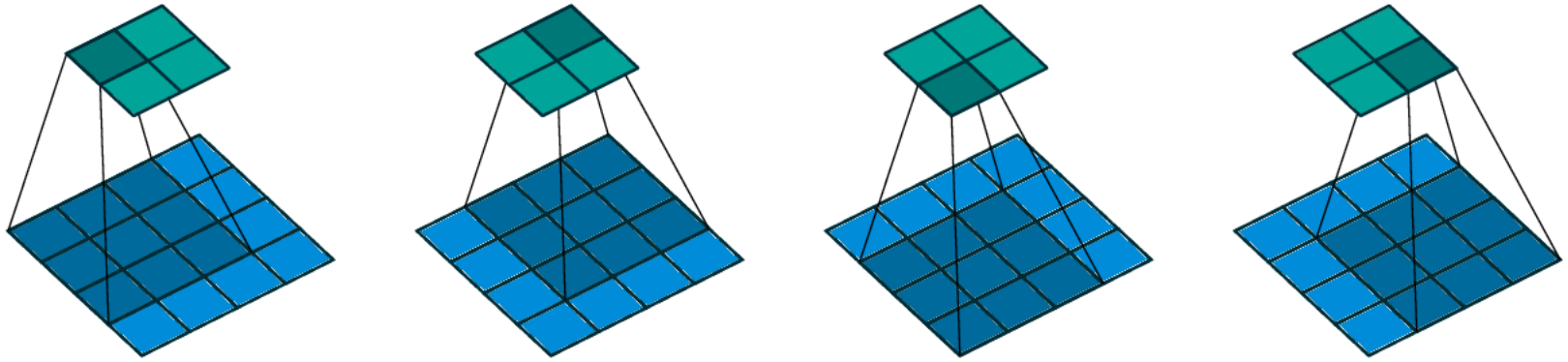
(i.e. 3*5*5 = 75-dimensional dot product + bias)

An example of convolving a 1x5x5 image with a 1x3x3 filter



Convolution layer

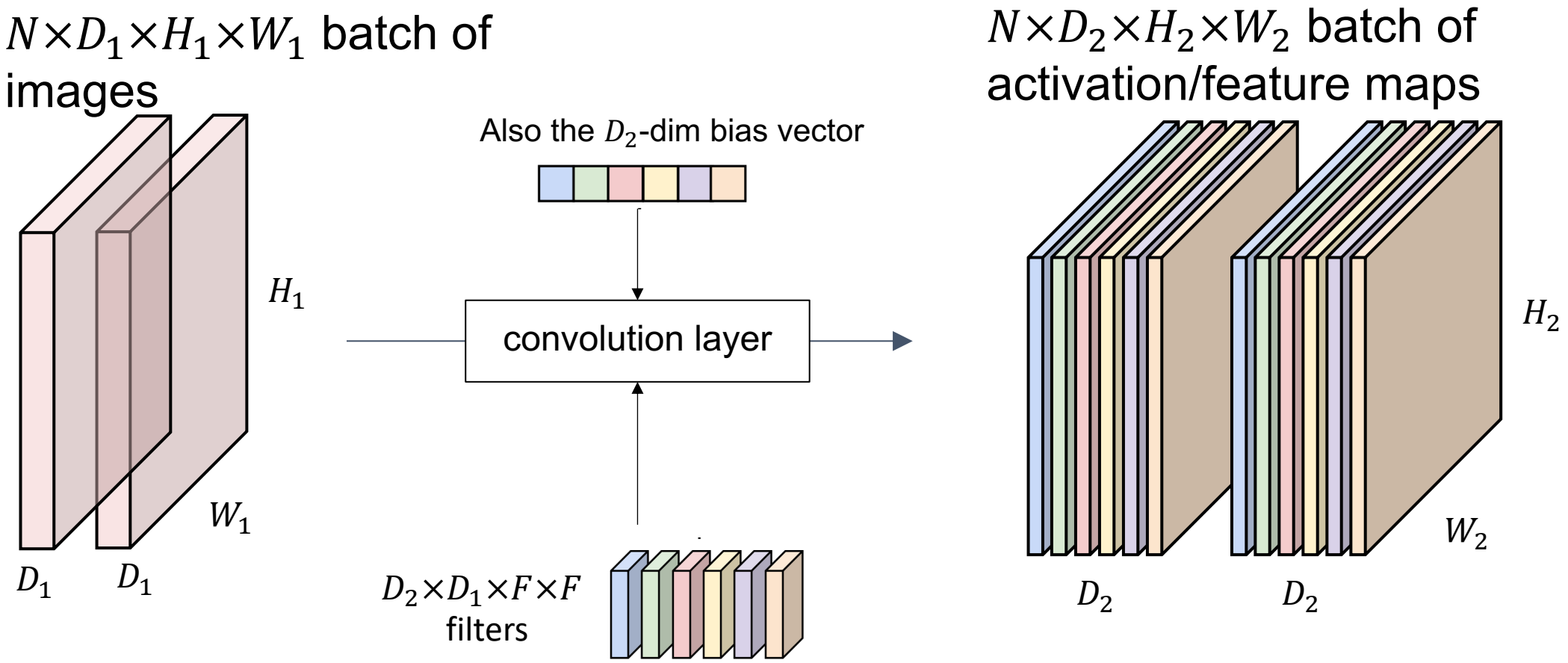
Convolution by doing a sliding window



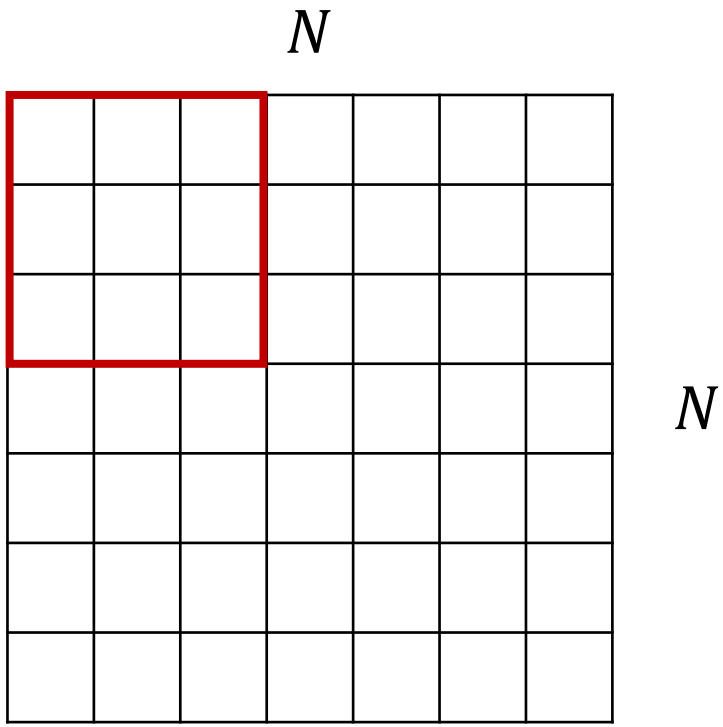
As a guiding example, let us consider the convolution of single-channel tensors $\mathbf{x} \in \mathbb{R}^{4\times4}$ and $\mathbf{w} \in \mathbb{R}^{3\times3}$:

$$\mathbf{w} \star \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

Convolution layer



Convolution layer – spatial dimensions



$N \times N$ input (spatially), assume $F \times F$ filter, and S stride

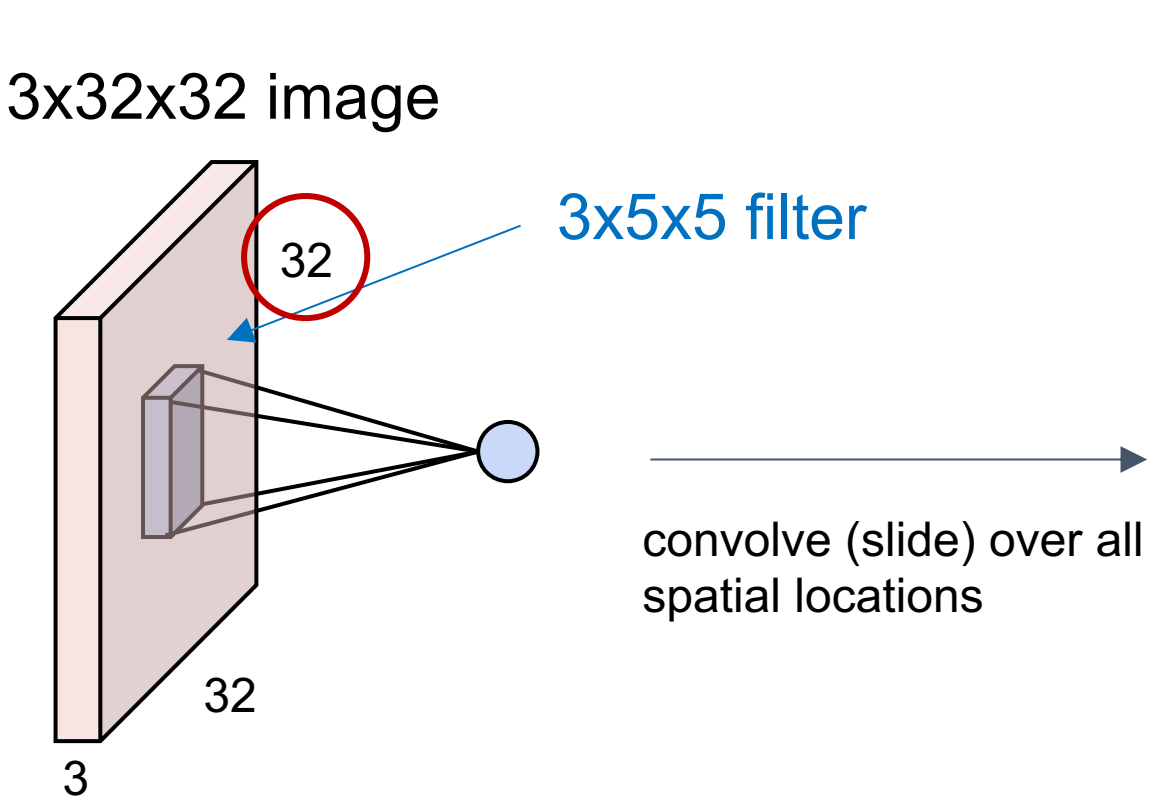
Output size = $\frac{N-F}{S} + 1$

e.g.

$$N = 7, F = 3$$

stride 1 => $(7 - 3)/1 + 1 = 5$
stride 2 => $(7 - 3)/2 + 1 = 3$
stride 3 => $(7 - 3)/3 + 1 = 2.33$:\

Convolution layer – spatial dimensions



Why does the feature map has a size of 28x28?

Output size = $\frac{N-F}{S} + 1$

$N = 32, F = 5$

stride 1 => $(32 - 5)/1 + 1 = 28$

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In PyTorch, by default, we pad top, bottom, left, right with zeros (this can be customized, e.g., 'constant', 'reflect', and 'replicate').

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output shape?

Recall that without padding, output size = $\frac{N-F}{S} + 1$

With padding, output size = $\frac{N-F+2P}{S} + 1$

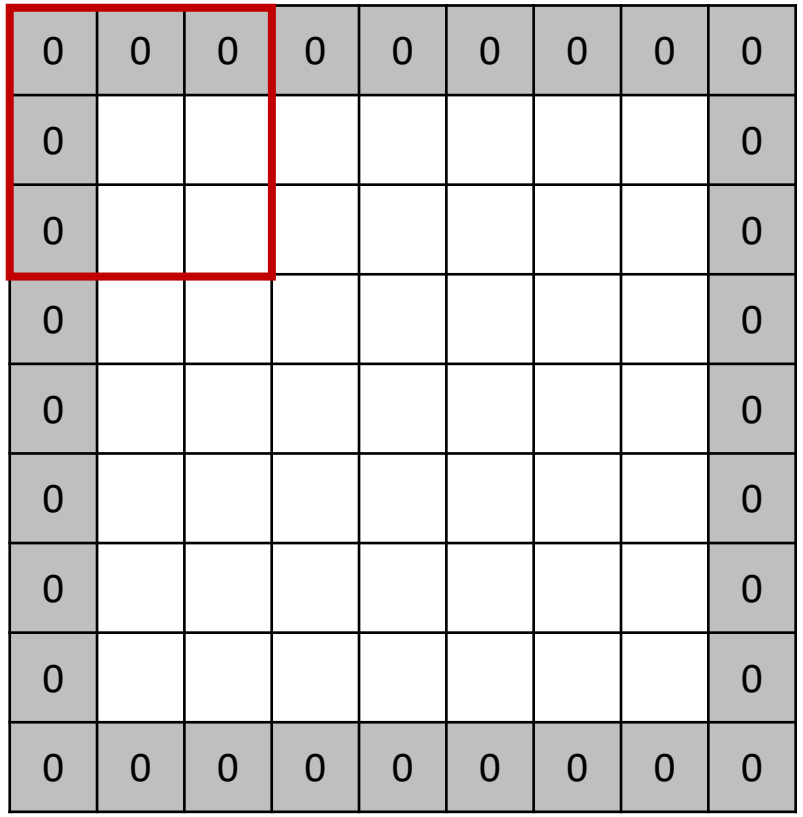
e.g.

$$N = 7, F = 3$$

stride 1 => $(7 - 3 + 2(1))/1 + 1 = 7$

Convolution layer – zero padding

By zero-padding in each layer, we prevent the representation from shrinking with depth. In practice, we zero pad the border. In PyTorch, by default, we pad top, bottom, left, right with zeros (this can be customized).



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output shape?

7×7 output!

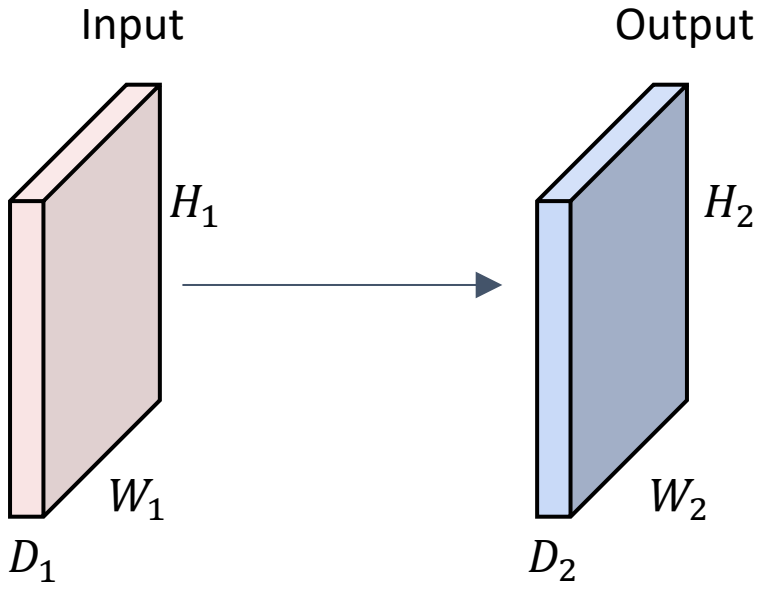
In general, common to see CONV layers with stride 1, filters of size $F \times F$, and zero-padding with $(F-1)/2$. (will preserve size spatially)

e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Convolution layer - summary

A convolution layer

- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires four hyperparameters
  - Number of filters K
  - Their spatial extent F
  - The stride S
  - The amount of zero padding P
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e., width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases
- In the output volume, the d-th depth slice (of size $H_2 \times W_2$) is the result of a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias



Pooling layer - summary

A pooling layer

- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires two hyperparameters
  - Their spatial extent F
  - The stride S
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for pooling layers

Outline – CNN I

- Basic components in CNN
- Training a classifier
- Optimizers

Outline – CNN II

- CNN Architectures

  You learn some classic architectures

- More on convolution
  - How to calculate FLOPs
  - Pointwise convolution
  - Depthwise convolution
  - Depthwise convolution + Pointwise convolution

  You learn how to calculate computation complexity of convolutional layer and how to design a lightweight network

- Batch normalization

  You learn an important technique to improve the training of modern neural networks

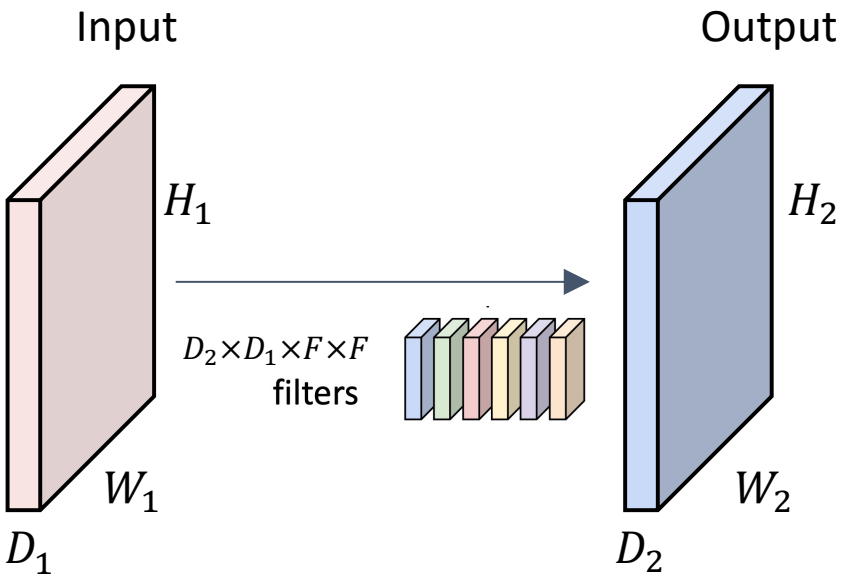
- Prevent overfitting
  - Transfer learning
  - Data augmentation

  You learn two important techniques to prevent overfitting in neural networks

How to calculate the computations of Convolution?

FLOPs (floating point operations)

Not FLOPS (floating point operations per second)



Assume:

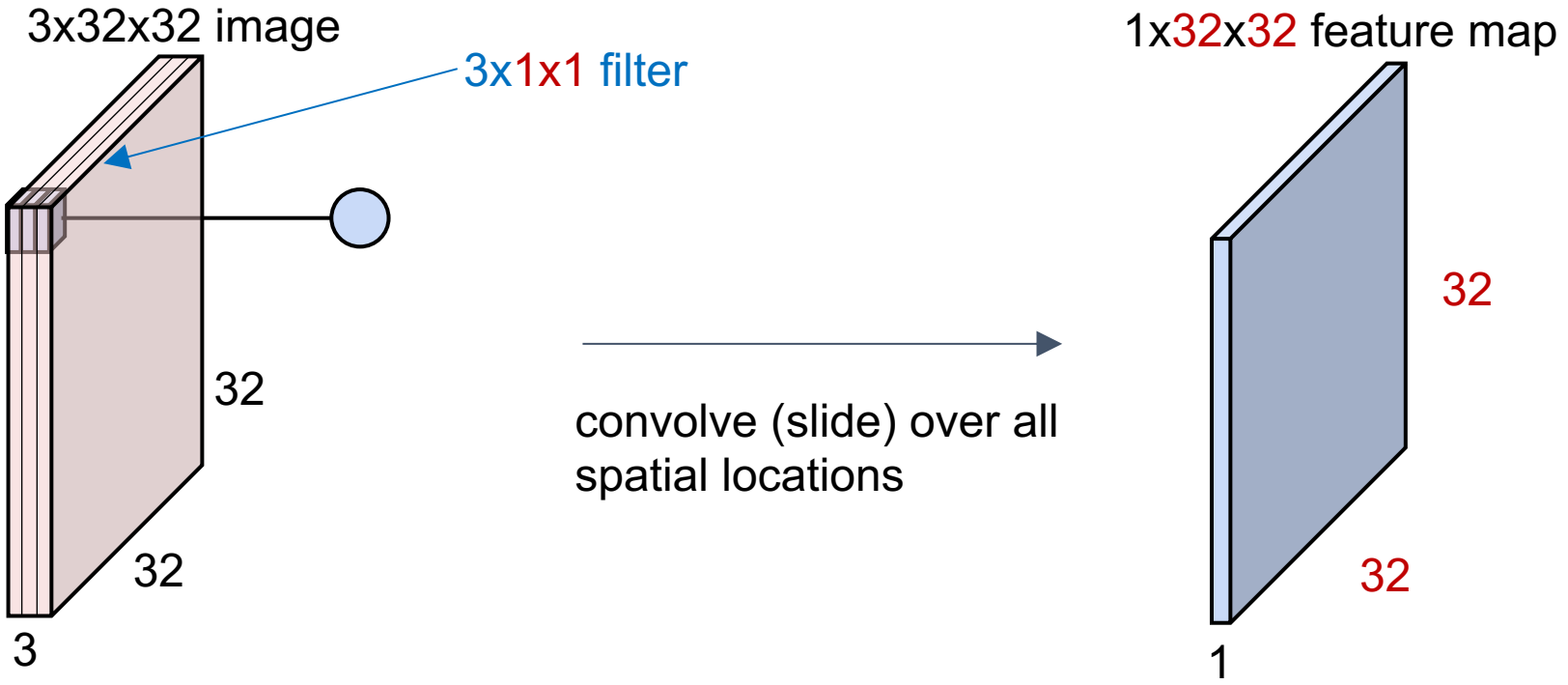
- Filter size F
- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Produces a output volume of size $D_2 \times H_2 \times W_2$

The FLOPs of the convolution layer is given by

$$\text{FLOPs} = \underbrace{[(D_1 \times F^2)}_{\text{Elementwise multiplication of each filter on a spatial location}} + \underbrace{(D_1 \times F^2 - 1)}_{\text{Adding the elements after multiplication, we need } n-1 \text{ adding operations for } n \text{ elements}} + \underbrace{1}_{\text{Add the bias}}] \times D_2 \times H_2 \times W_2 = (2 \times D_1 \times F^2) \times D_2 \times H_2 \times W_2$$

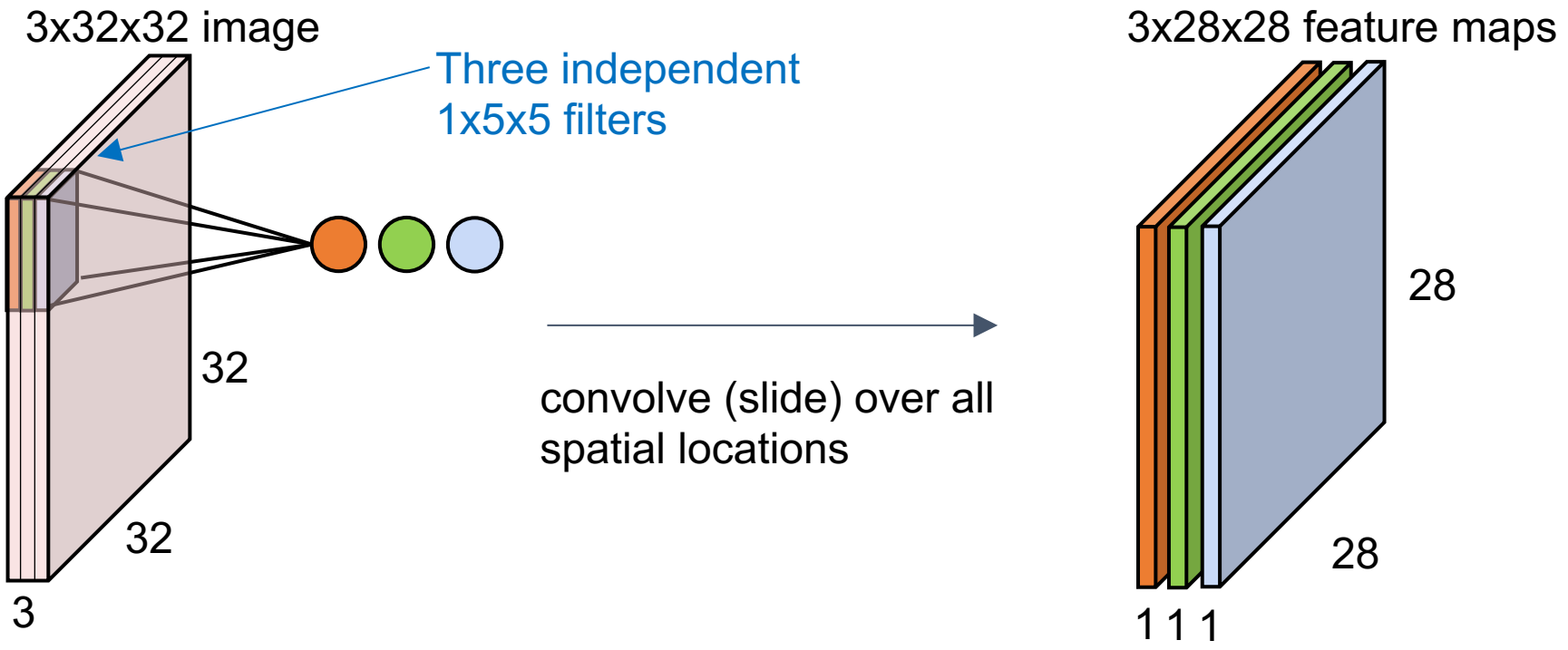
Pointwise convolution

- Why having filter of spatial size 1×1?
  - Change the size of channels
  - "Blend" information among channels by linear combination



Depthwise convolution

- Depthwise convolution
  - Convolution is performed independently for each of input channels



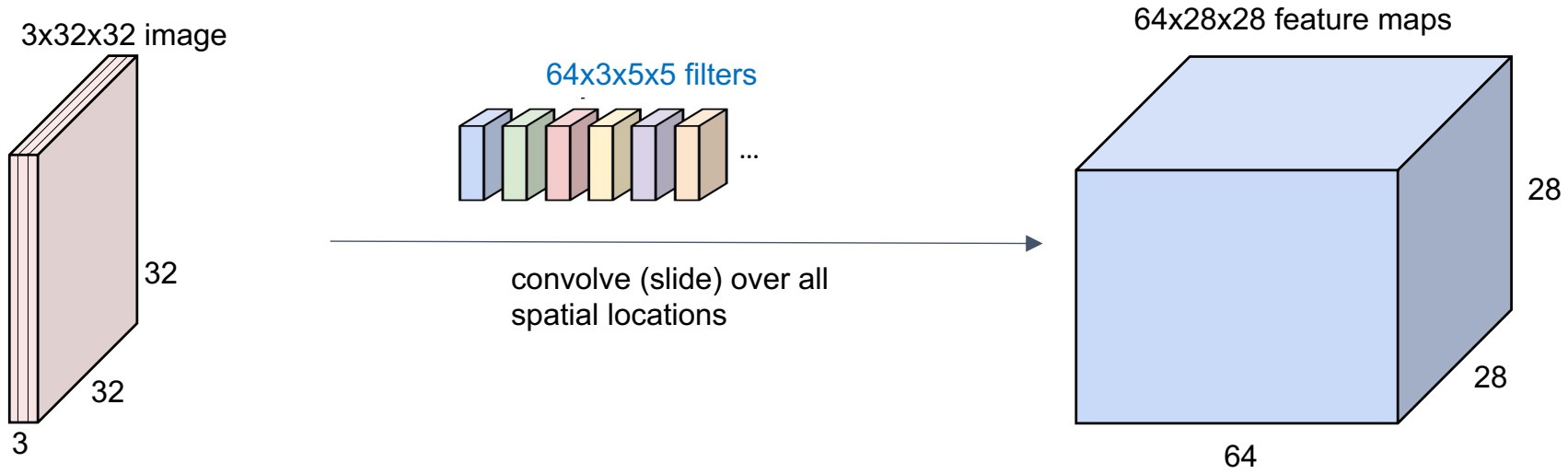
Depthwise convolution + Pointwise convolution

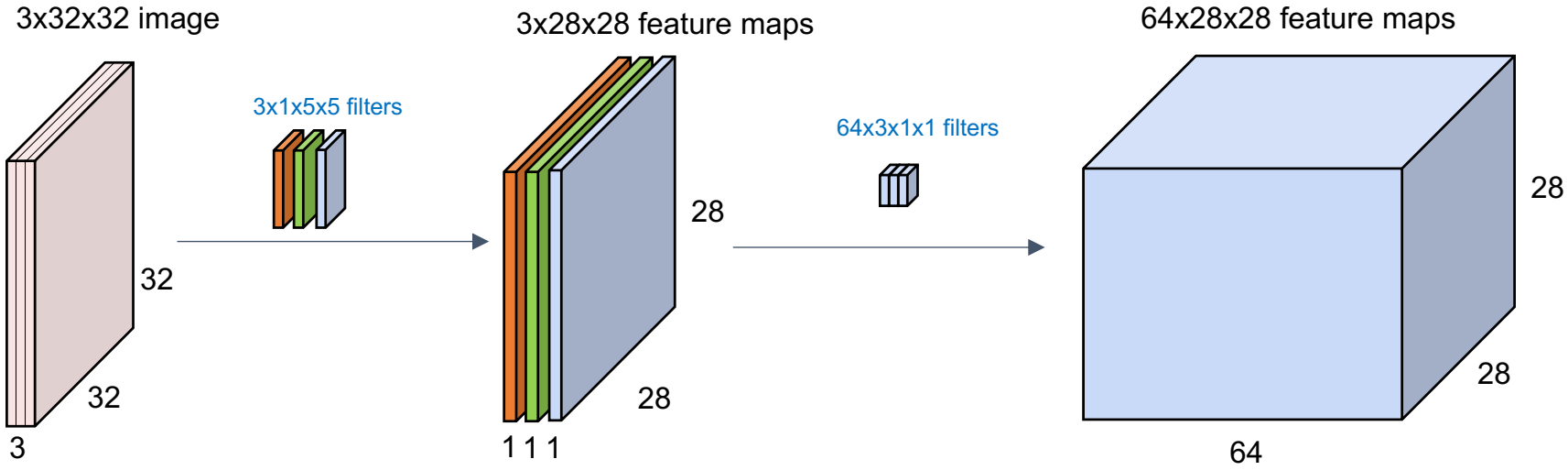
Replace standard convolution

with

Depthwise convolution + pointwise convolution

And we still get the same size of output volume!





Depthwise convolution + Pointwise convolution

- How much computation do you save by replacing standard convolution with depthwise+pointwise?

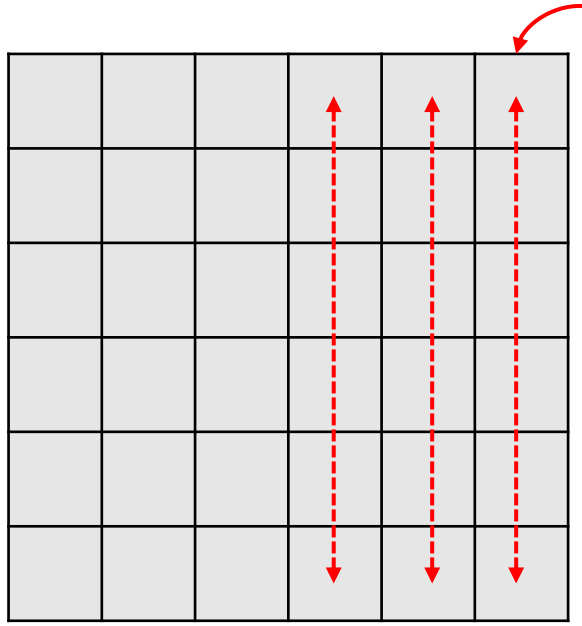
$$\text{Ratio} = \frac{\text{Cost of depthwise convolution} + \text{pointwise convolution}}{\text{Cost of standard convolution}}$$

$$\text{Ratio} = \frac{(2\times F^2) \times D_1 \times H_2 \times W_2}{(2\times D_1 \times F^2) \times D_2 \times H_2 \times W_2} + \frac{(2\times D_1) \times D_2 \times H_2 \times W_2}{(2\times D_1 \times F^2) \times D_2 \times H_2 \times W_2}$$

$$\text{Ratio} = \frac{1}{D_2} + \frac{1}{F^2}$$

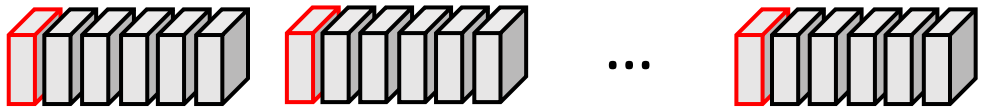
D_2 is usually large. Reduction rate is roughly 1/8–1/9 if 3×3 depthwise separable convolutions are used

Batch Normalization

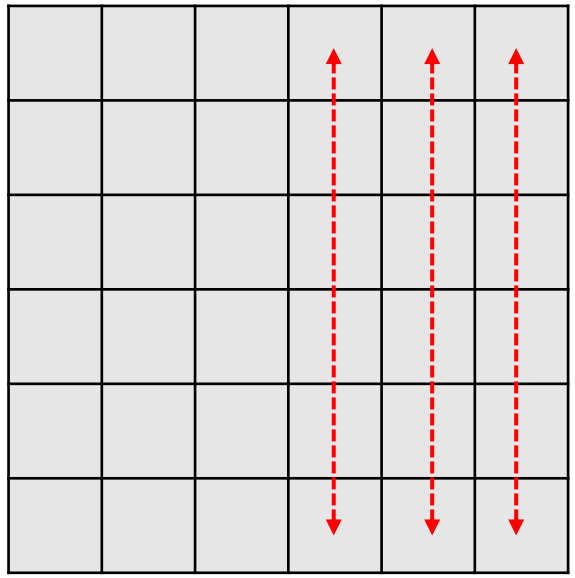


The goal of batch normalization is to normalize the values across each column so that the values of the column have zero mean and unit variance

For instance, normalizing the first column of this matrix means normalizing the activations (highlighted in red) below



Batch Normalization - Training



$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the **normalized** values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β, shape is $1 \times D$

Batch Normalization – Test Time

Input

Channel, D

Problem: Estimates depend on minibatch; can't do this at test-time

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is $1 \times D$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β, shape is $1 \times D$

Batch Normalization – Test Time

Input

Channel, D

Average of values seen during training

Per-channel mean, shape is $1 \times D$

Average of values seen during training

Per-channel variance, shape is $1 \times D$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalize the values, shape is $N \times D$

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Scale and shift the normalized values to add flexibility, output shape is $N \times D$

Learnable parameters: γ and β, shape is $1 \times D$

Outline – CNN II

- CNN Architectures

  You learn some classic architectures

- More on convolution
  - How to calculate FLOPs
  - Pointwise convolution
  - Depthwise convolution
  - Depthwise convolution + Pointwise convolution

  You learn how to calculate computation complexity of convolutional layer and how to design a lightweight network

- Batch normalization

  You learn an important technique to improve the training of modern neural networks

- Prevent overfitting
  - Transfer learning
  - Data augmentation

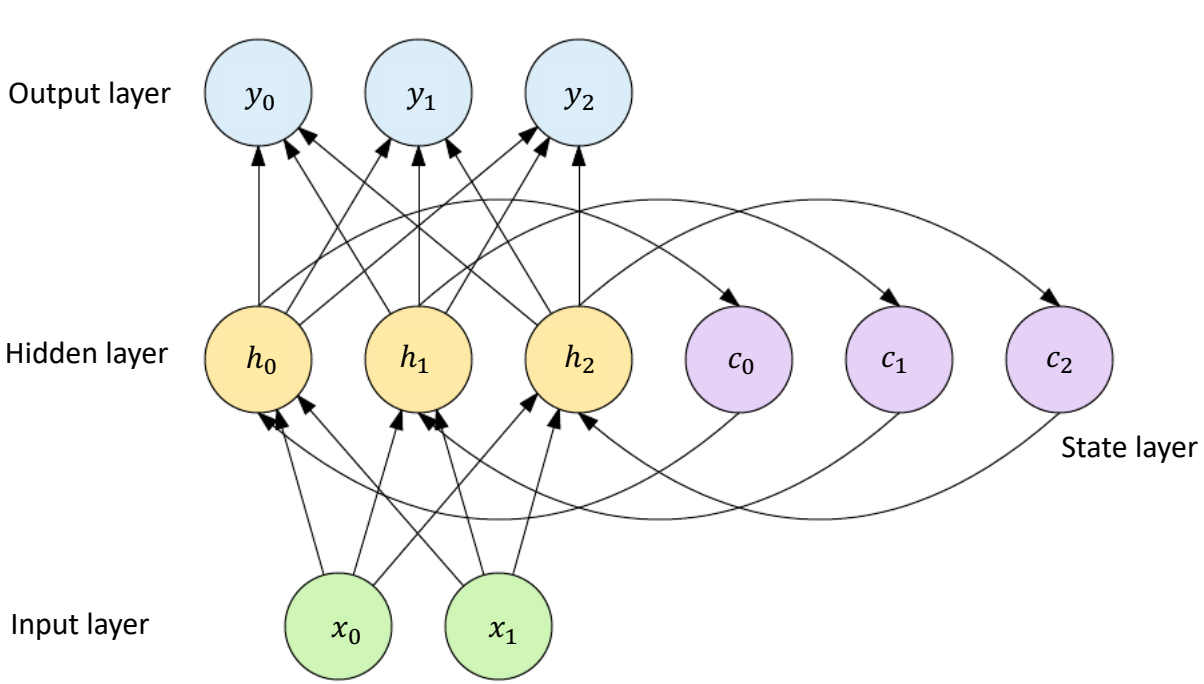
  You learn two important techniques to prevent overfitting in neural networks

Outline – RNN

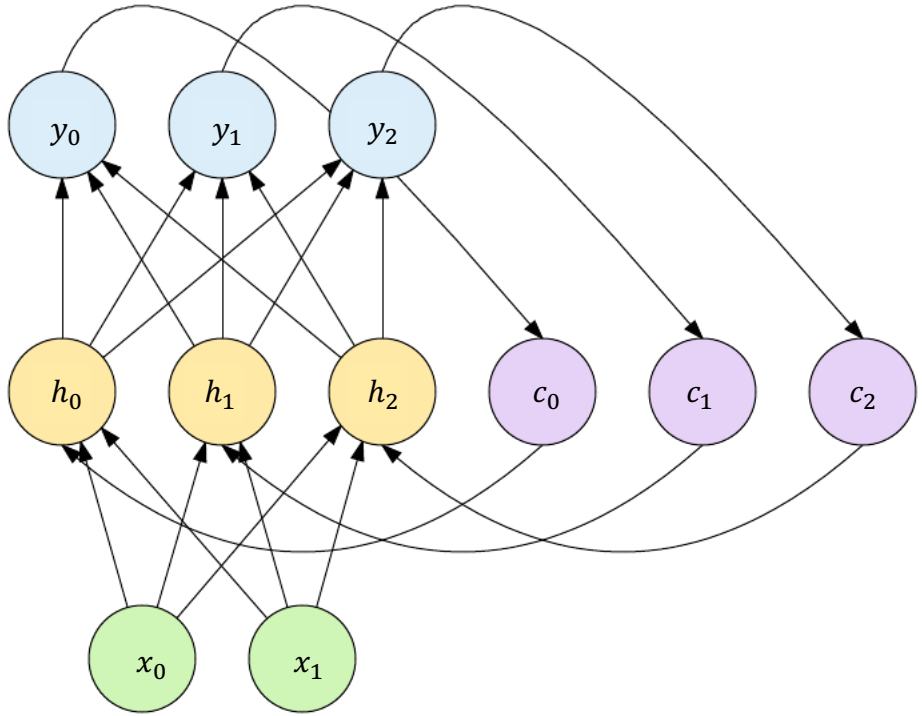
- Recurrent Neural Network (RNN)
  - Hidden recurrence
  - Top-down recurrence
- Long Short-Term Memory (LSTM)
  - Long-term dependency
  - Structure of LSTM
- Example Applications

Types of RNN

RNN with hidden recurrence
(Elman-type)



RNN with top-down recurrence
(Jordan-type)

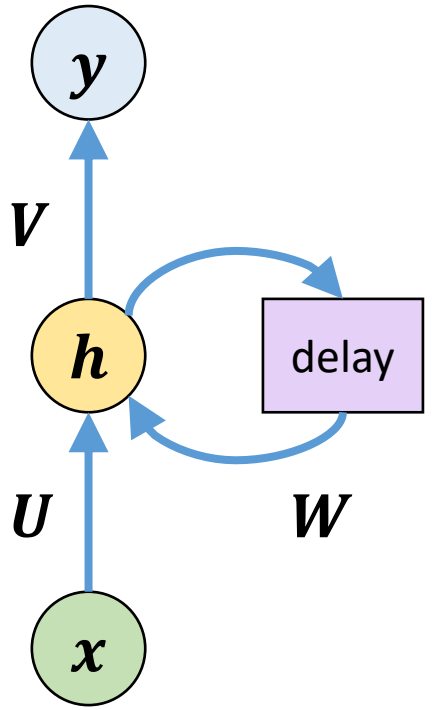


RNN with hidden recurrence

Let $\boldsymbol{x}(t)$, $\boldsymbol{y}(t)$, and $\boldsymbol{h}(t)$ be the input, output, and hidden output of the network at time t.

Activation of the Elman-type RNN with one hidden-layer is given by:

$$\boldsymbol{h}(t) = \phi\left(\boldsymbol{U}^{\mathsf{T}}\boldsymbol{x}(t) + \boldsymbol{W}^{\mathsf{T}}\boldsymbol{h}(t-1) + \boldsymbol{b}\right)$$
$$\boldsymbol{y}(t) = \sigma\left(\boldsymbol{V}^{\mathsf{T}}\boldsymbol{h}(t) + \boldsymbol{c}\right)$$



σ is a *softmax* function for classification and a *linear* function for regression.

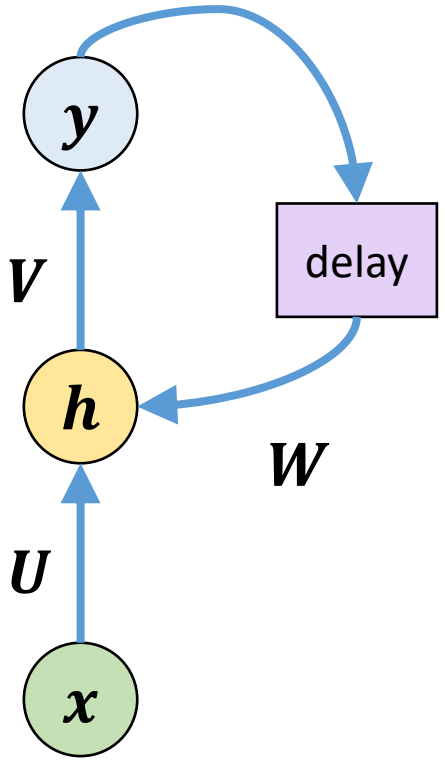
RNN with top-down recurrence

Let $\boldsymbol{x}(t)$, $\boldsymbol{y}(t)$, and $\boldsymbol{h}(t)$ be the input, output, and hidden output of the network at time t.

Activation of the Jordan-type RNN with one hidden-layer is given by:

$$\boldsymbol{h}(t) = \phi\left(\boldsymbol{U}^{\mathsf{T}}\boldsymbol{x}(t) + \boldsymbol{W}^{\mathsf{T}}\boldsymbol{y}(t-1) + \boldsymbol{b}\right)$$
$$\boldsymbol{y}(t) = \sigma\left(\boldsymbol{V}^{\mathsf{T}}\boldsymbol{h}(t) + \boldsymbol{c}\right)$$

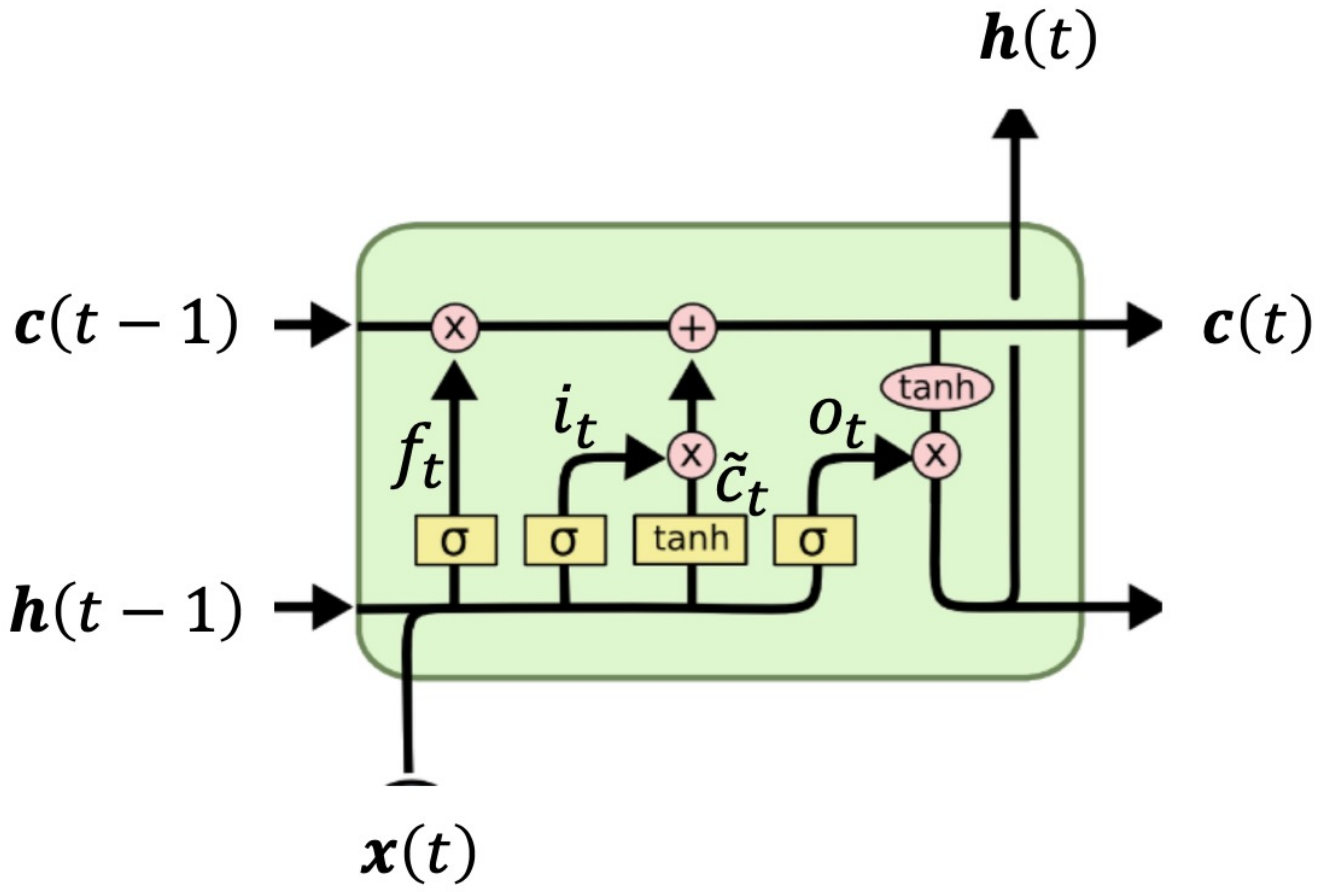
Note that output of the previous time instant is fed back to the hidden layer and $\boldsymbol{W}$ represents the recurrent weight matrix connecting previous output to the current hidden input



Long short-term memory (LSTM) unit

LSTMs provide a solution by incorporating memory units that allow the network to learn when to forget previous hidden states and when to update hidden states given new information.

Instead of having a single neural network layer, there are four, interacting in a very special way.



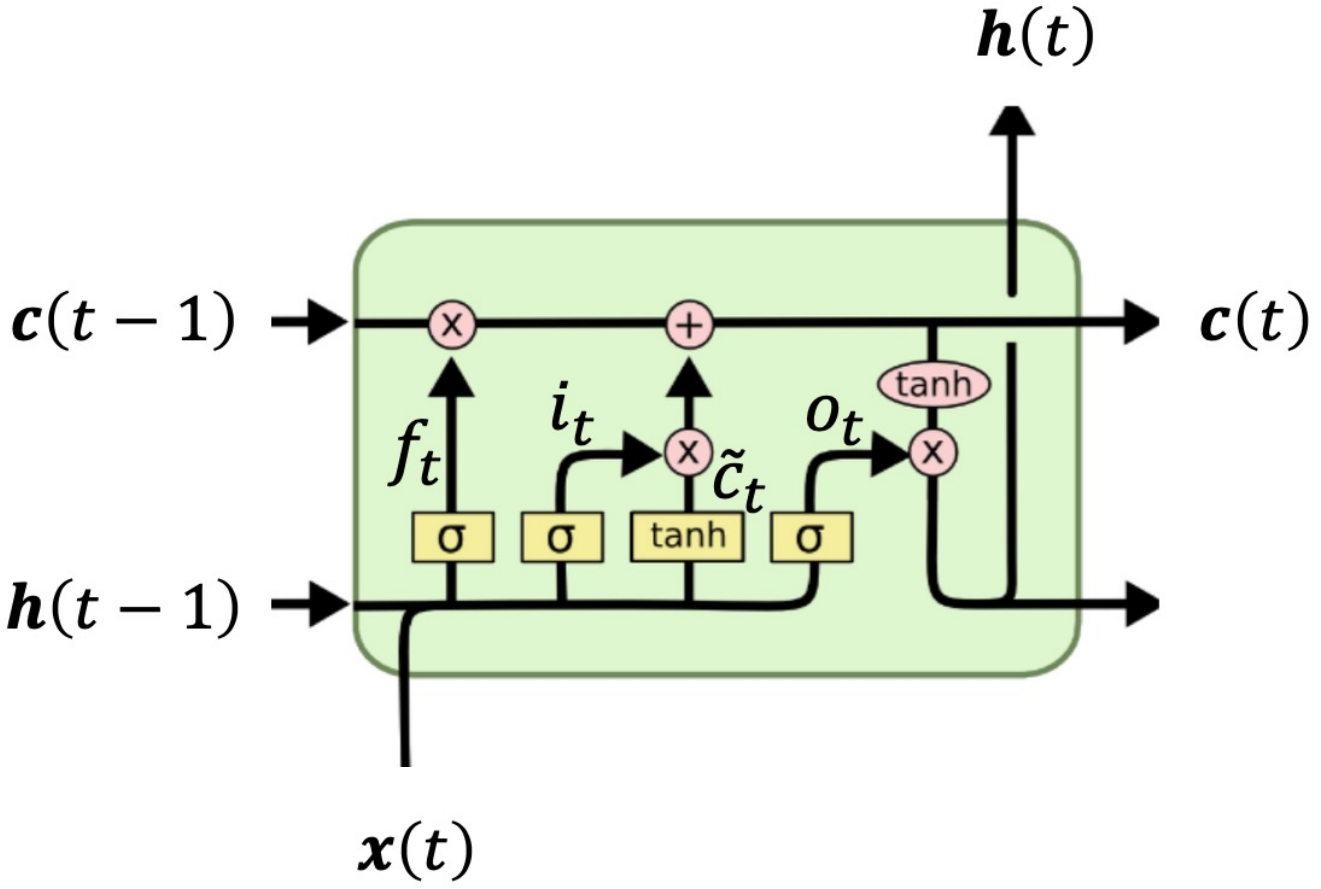
Hochreiter & Schmidhuber (1997)

LSTM unit

$$\boldsymbol{i}(t) = \sigma\left(\boldsymbol{U}_{\boldsymbol{i}}^{\top}\boldsymbol{x}(t) + \boldsymbol{W}_{\boldsymbol{i}}^{\top}\boldsymbol{h}(t-1) + \boldsymbol{b}_{\boldsymbol{i}}\right)$$
$$\boldsymbol{f}(t) = \sigma\left(\boldsymbol{U}_{\boldsymbol{f}}^{\top}\boldsymbol{x}(t) + \boldsymbol{W}_{\boldsymbol{f}}^{\top}\boldsymbol{h}(t-1) + \boldsymbol{b}_{\boldsymbol{f}}\right)$$
$$\boldsymbol{o}(t) = \sigma\left(\boldsymbol{U}_{\boldsymbol{o}}^{\top}\boldsymbol{x}(t) + \boldsymbol{W}_{\boldsymbol{o}}^{\top}\boldsymbol{h}(t-1) + \boldsymbol{b}_{\boldsymbol{o}}\right)$$

$$\tilde{\boldsymbol{c}}(t) = \phi\left(\boldsymbol{U}_{\boldsymbol{c}}^{\top}\boldsymbol{x}(t) + \boldsymbol{W}_{\boldsymbol{c}}^{\top}\boldsymbol{h}(t-1) + \boldsymbol{b}_{\boldsymbol{c}}\right)$$

$$\boldsymbol{c}(t) = \tilde{\boldsymbol{c}}(t) \odot \boldsymbol{i}(t) + \boldsymbol{c}(t-1) \odot \boldsymbol{f}(t)$$
$$\boldsymbol{h}(t) = \phi\left(\boldsymbol{c}(t)\right) \odot \boldsymbol{o}(t)$$



Outline – Attention

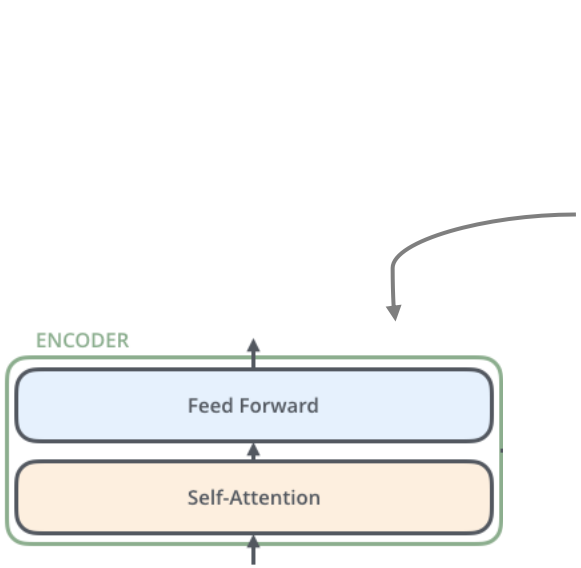
- Attention
- Transformers
- Vision Transformers

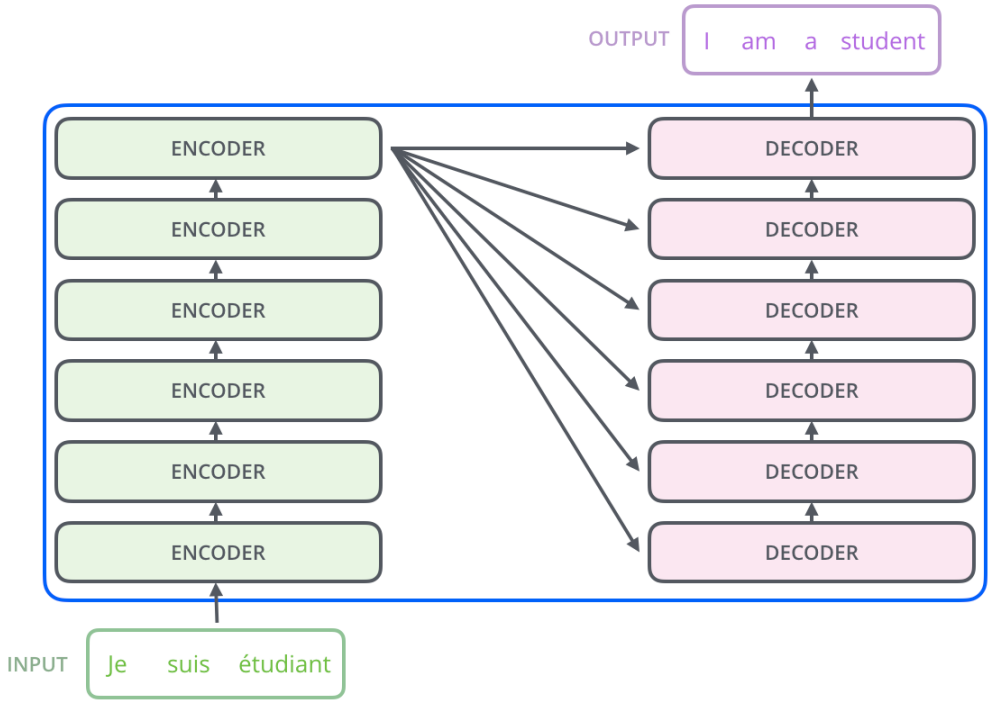
Transformers

The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word

The outputs of the self-attention layer are fed to a feed-forward neural network.

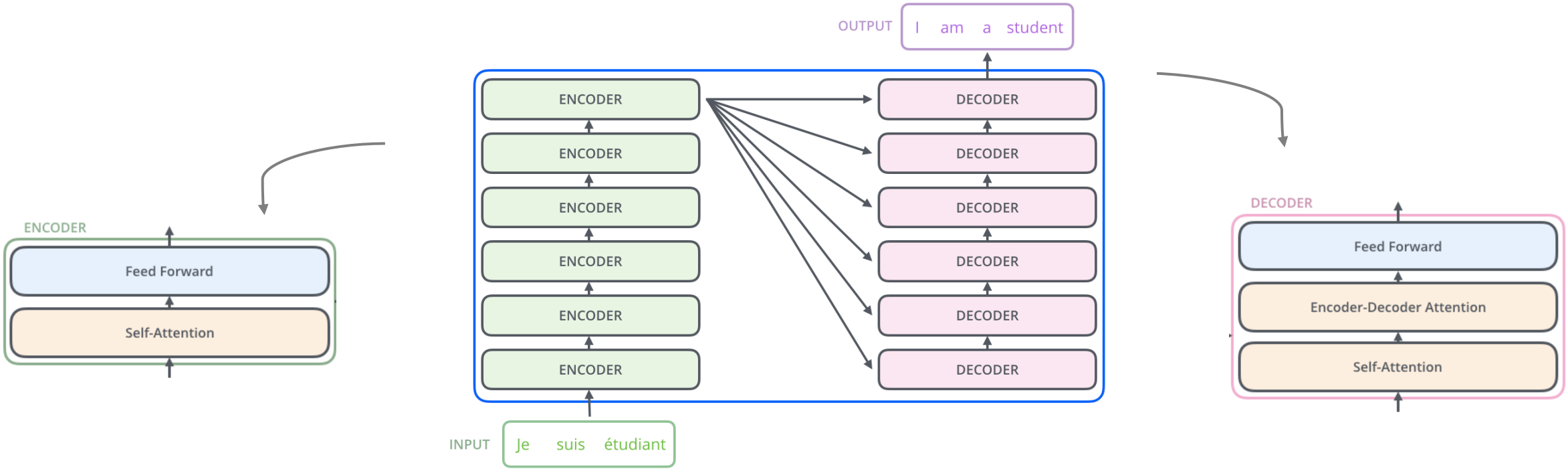
The exact same feed-forward network is independently applied to each position (each word/token).



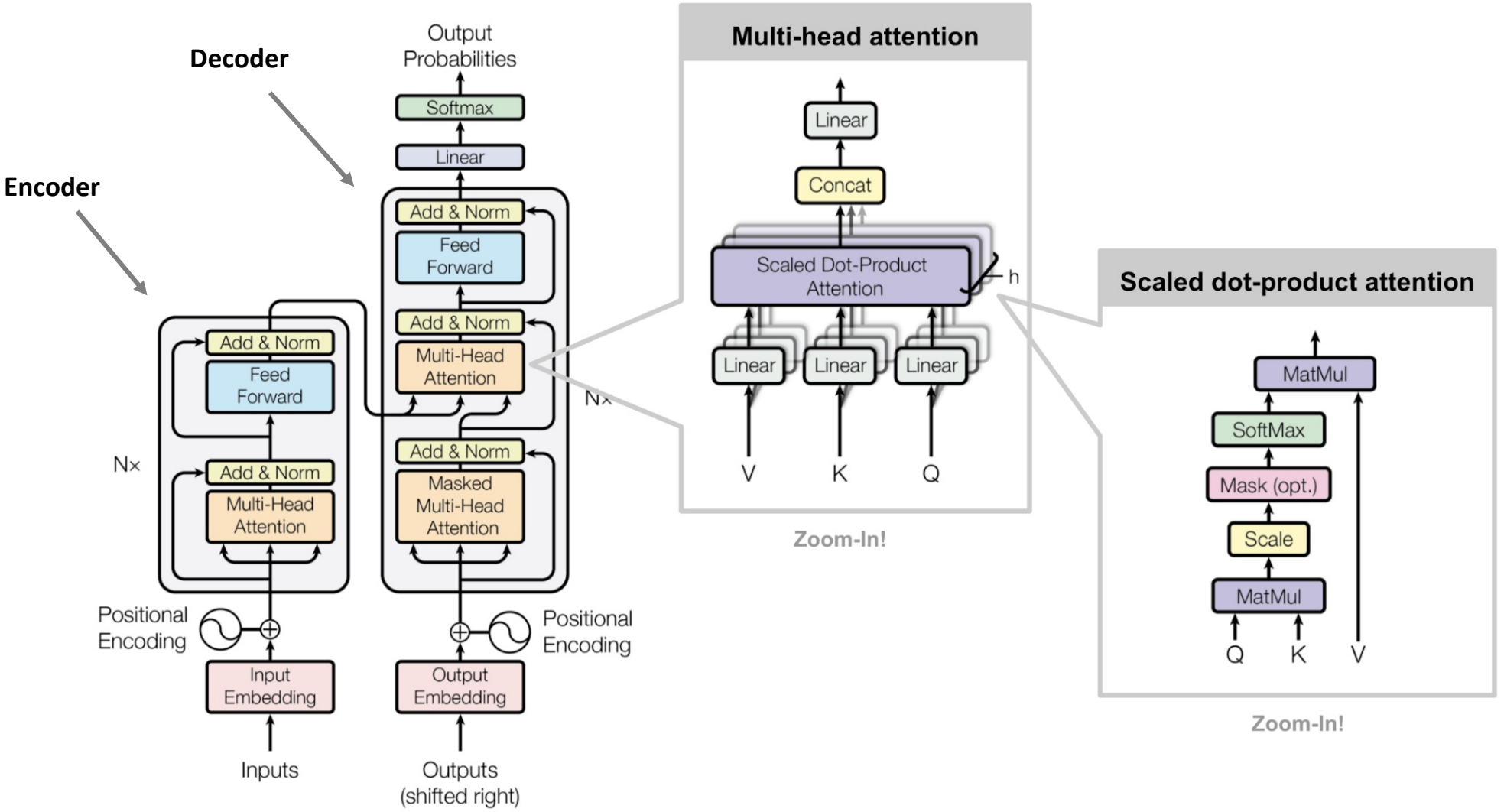


Transformers

The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence



Transformers

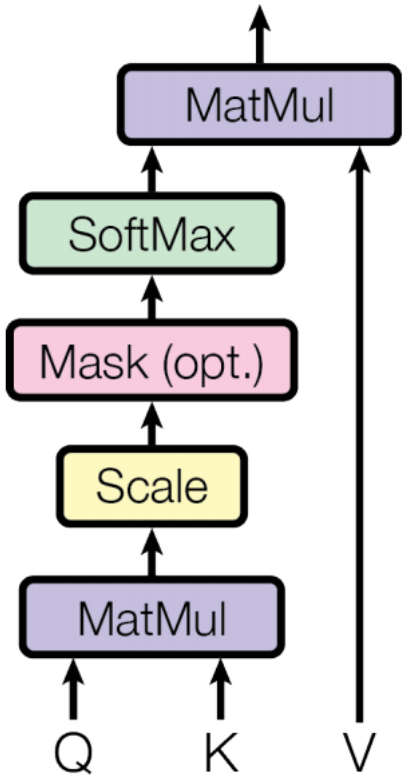


Transformers

Self-attention = Scaled dot-product attention

The output is a weighted sum of the values, where the weight assigned to each value is determined by the dot-product of the query with all the keys

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$



Scaled Dot-Product Attention

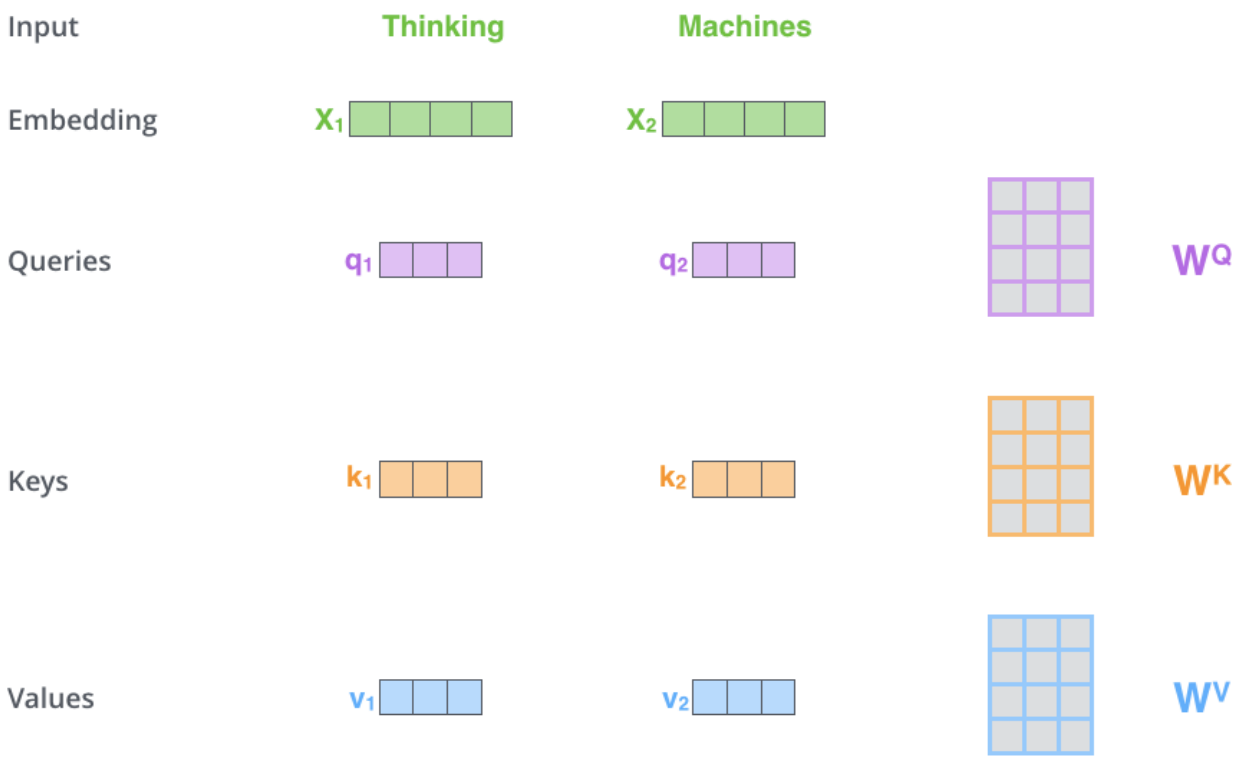
Self-Attention in Detail

First Step

Create three vectors from each of the encoder's input vectors (in this case, the **embedding** of each word).

So for each word, we create a **Query vector**, a **Key vector**, and a **Value vector**.

These vectors are created by multiplying the embedding by three matrices that we trained during the training process.



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Self-Attention in Detail

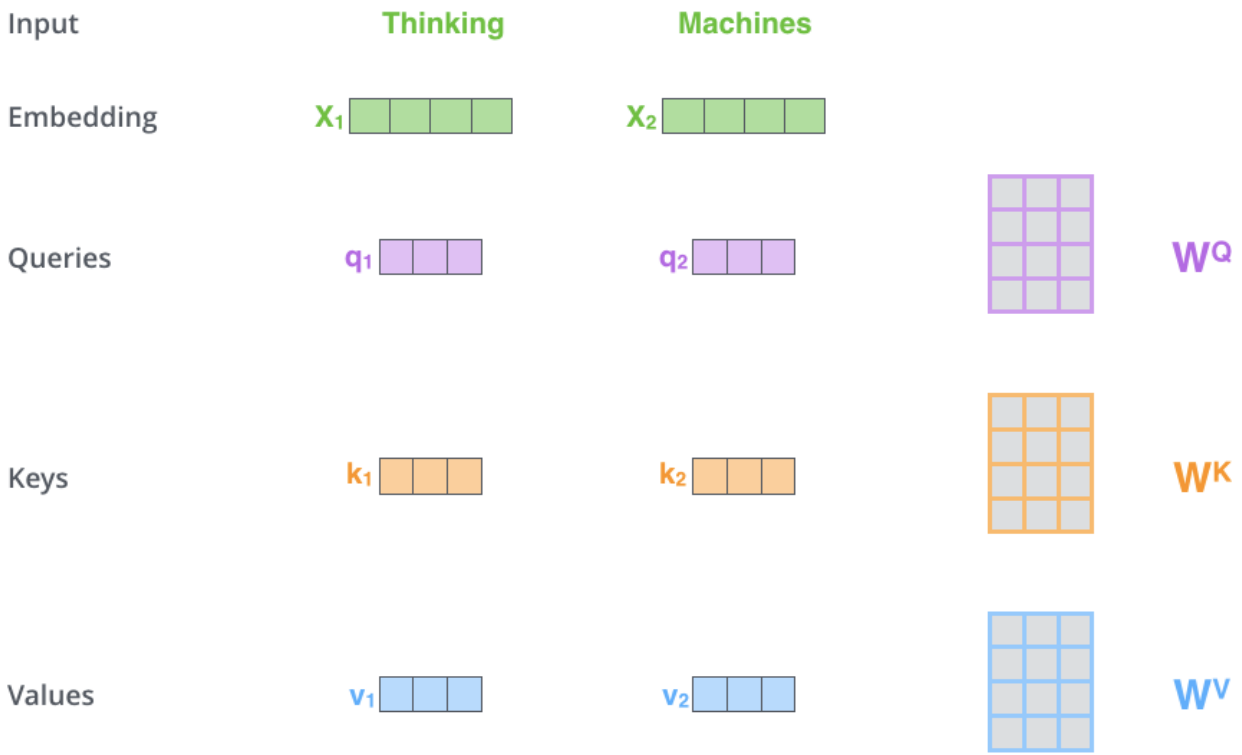
First Step

What are the "query", "key", and "value" vectors?

The names *"query", "key"* are inherited from the field of information retrieval

The dot product operation returns a measure of similarity between its inputs, so the weights $\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{d_k}}$ depend on the relative similarities between the n-th query and all of the keys

The softmax function means that the key vectors "compete" with one another to contribute to the final result.



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{d_k}}\right)\mathbf{V}$$

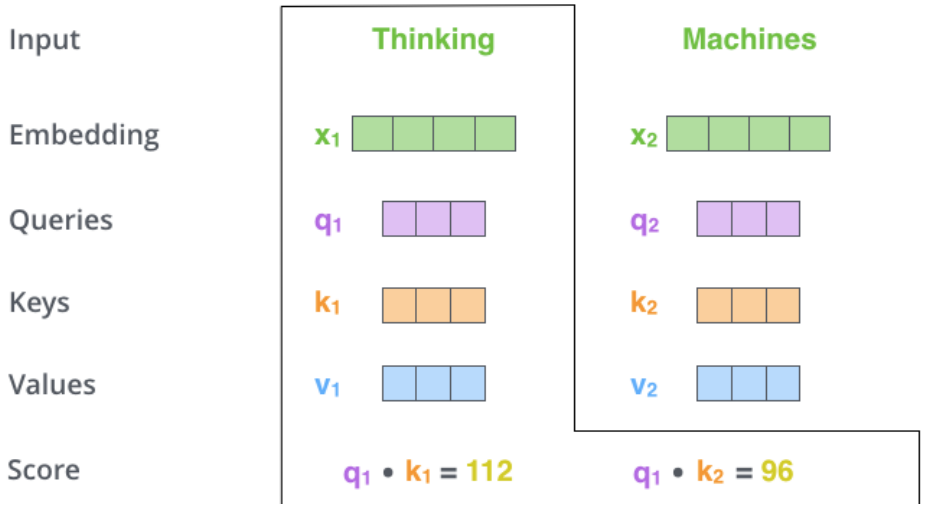
Self-Attention in Detail

Second Step

Calculate a score for each word of the input sentence against a word.

The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring.



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Self-Attention in Detail

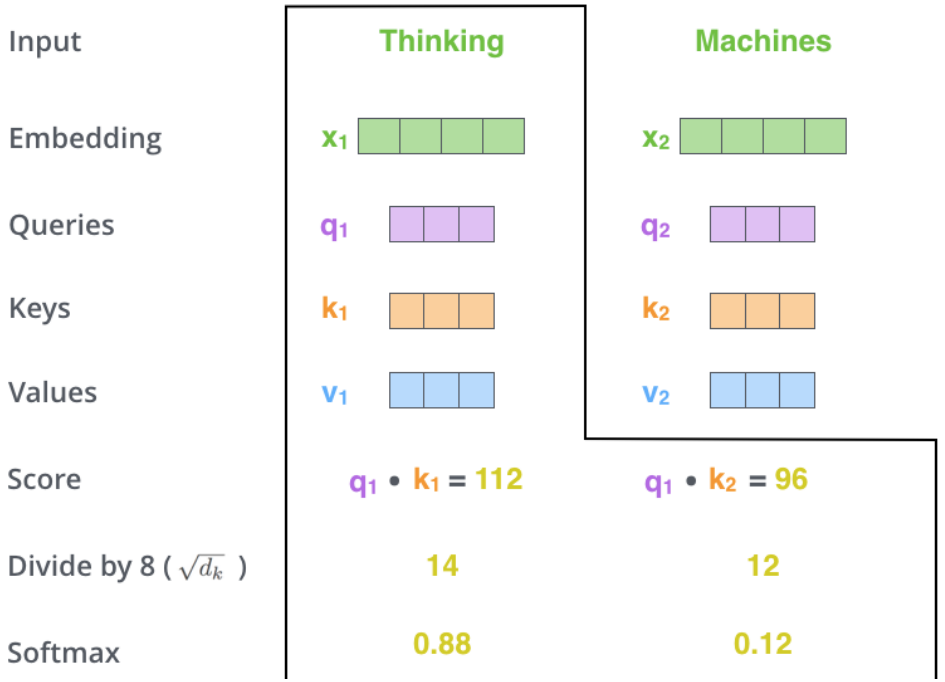
Third Step

Divide the scores by $\sqrt{d_k}$, the square root of the dimension of the key vectors

This leads to having more stable gradients (large similarities will cause softmax to saturate and give vanishing gradients)

Fourth Step

Softmax for normalization



$$z_1 = q_1 . k_1 / \sqrt{d_k} = 112/\sqrt{64} = 112/8 = 14, z_2 = q_1 . k_2 / \sqrt{d_k} = 96/\sqrt{64} = 96/8 = 12$$

$$softmax(z_1) = exp(z_1) / \sum_{i=1}^{2}(exp(z_i)) = 0.88, softmax(z_2) = exp(z_2) / \sum_{i=1}^{2}(exp(z_i)) = 0.12$$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

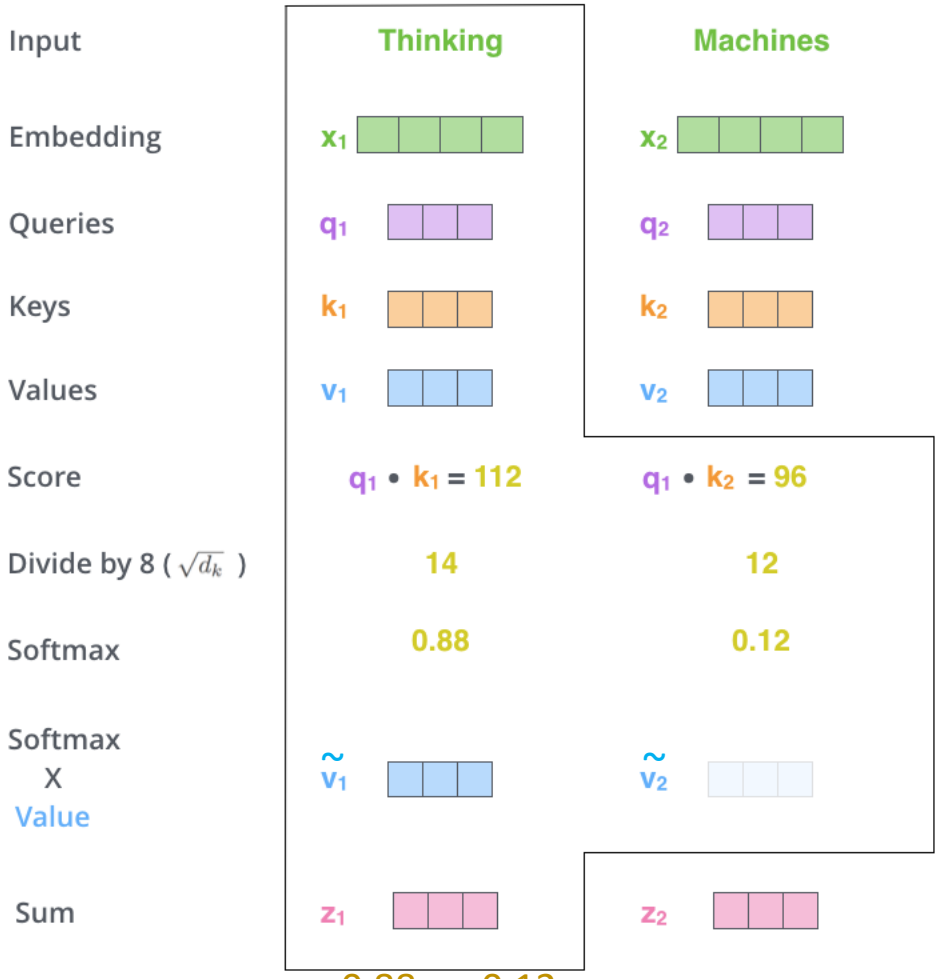
Self-Attention in Detail

Fifth Step

Multiply each value vector by the softmax score

Sixth Step

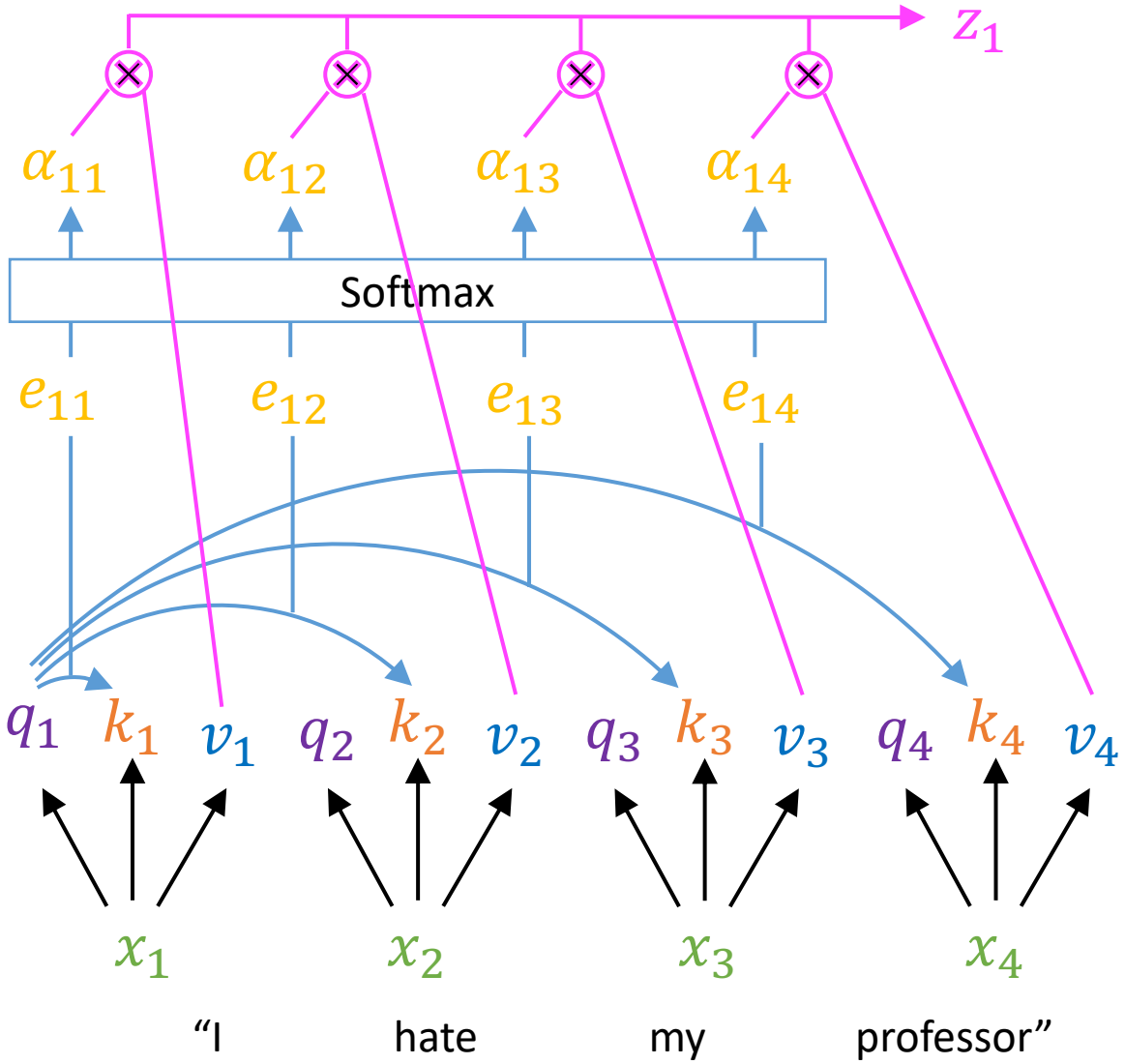
Sum up the weighted value vectors to get the output of the self-attention layer at this position (for the first word)



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{SoftMax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Self-Attention in Detail

$$e = \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}$$



$$q = W_q x$$

$$k = W_k x$$

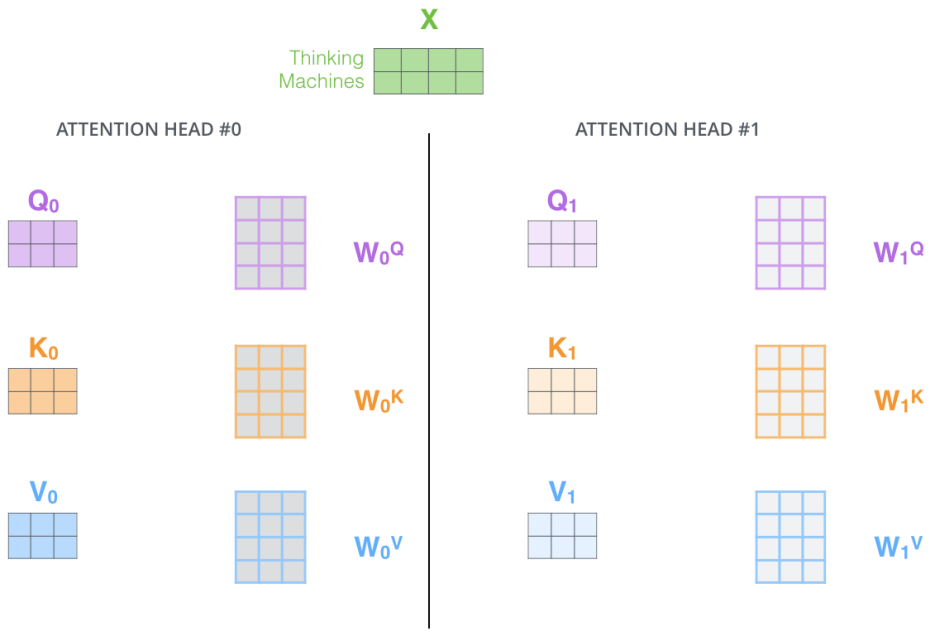
$$v = W_v x$$

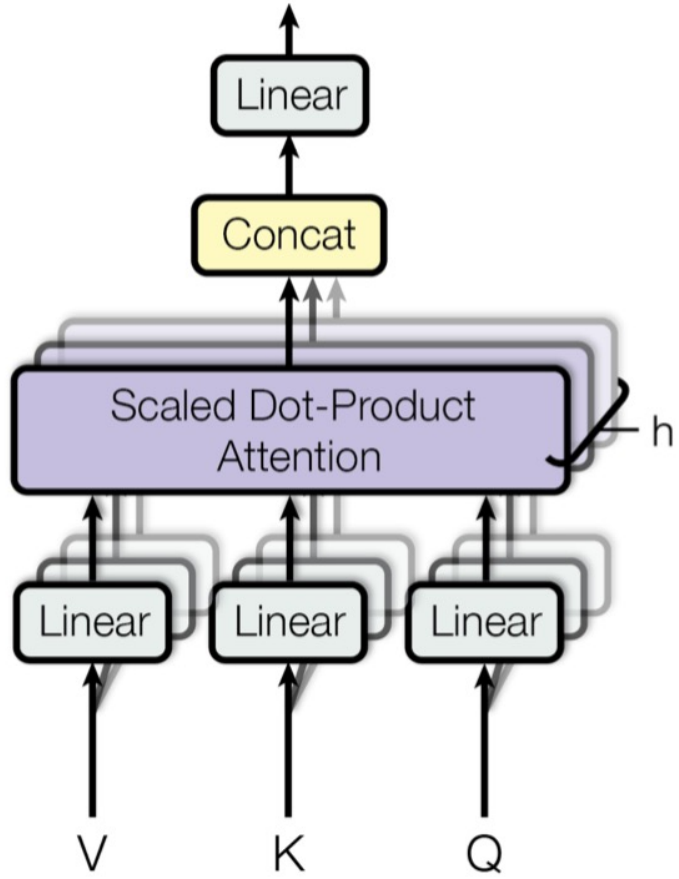
Multi-Head Self-Attention

Multi-Head Self-Attention

Rather than only computing the attention once, the multi-head mechanism runs through the scaled dot-product attention multiple times in parallel.

Due to different linear mappings, each head is presented with different versions of keys, queries, and values





Multi-head Attention

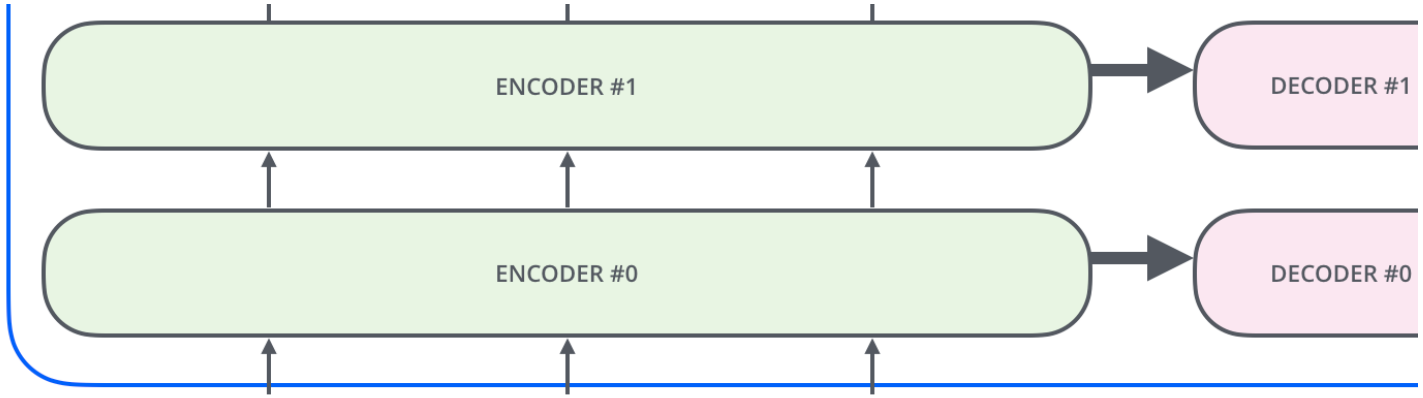
Positional Encoding

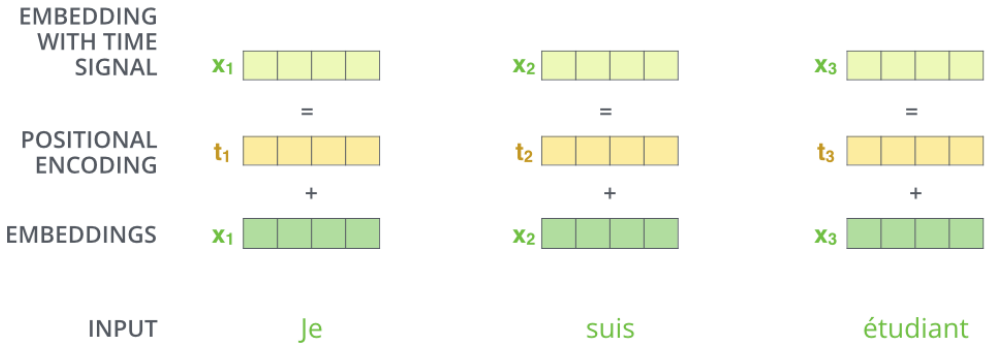
Self-attention layer works on sets of vectors and it doesn't know the order of the vectors it is processing

The positional encoding has the same dimension as the input embedding

Adds a vector to each input embedding to give information about the relative or absolute position of the tokens in the sequence

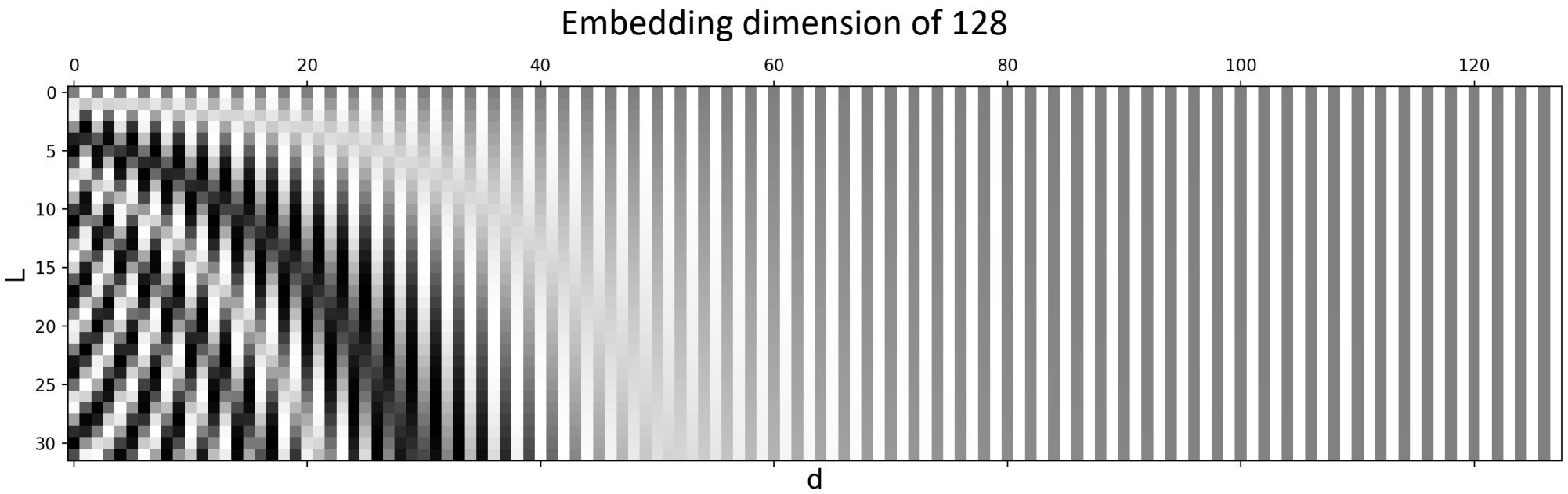
These vectors follow a specific pattern





Positional Encoding

32 tokens



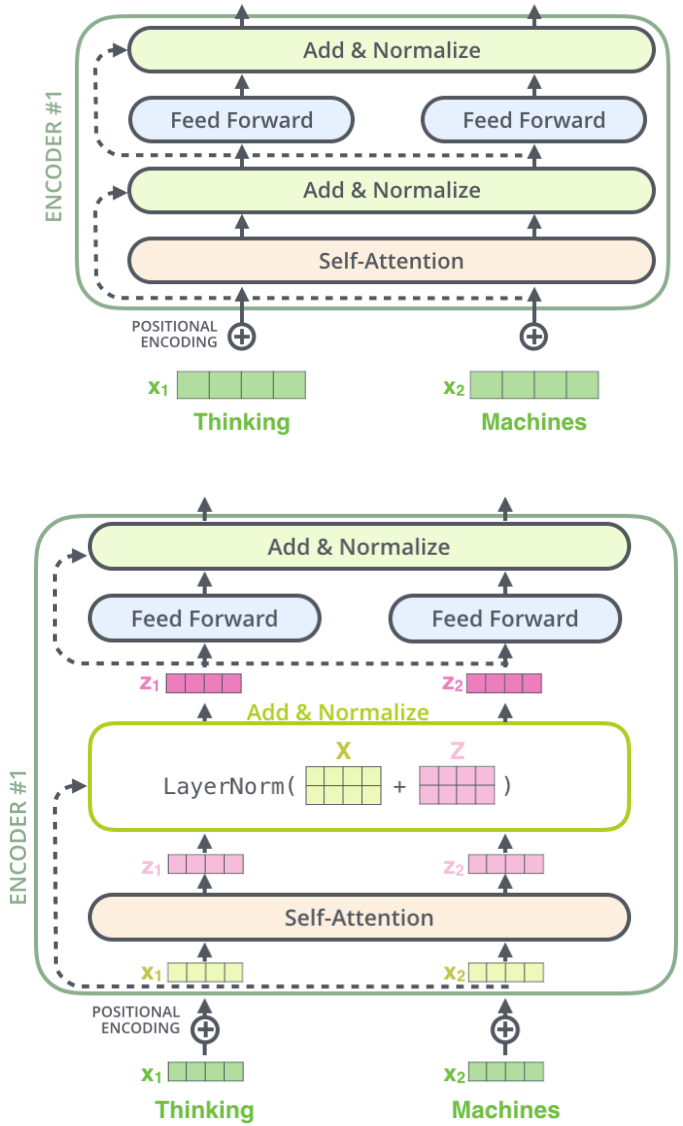
*Sinusoidal positional encoding - interweaves the two signals (*sine for even indices and cosine for odd indice*)*

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

where pos is the position and i is the dimension, $[0, \ldots, d_{model}/2\ )$

Transformer Encoder



If we're to visualize the vectors and the layer-norm operation associated with self attention, it would look like this:

Encoder

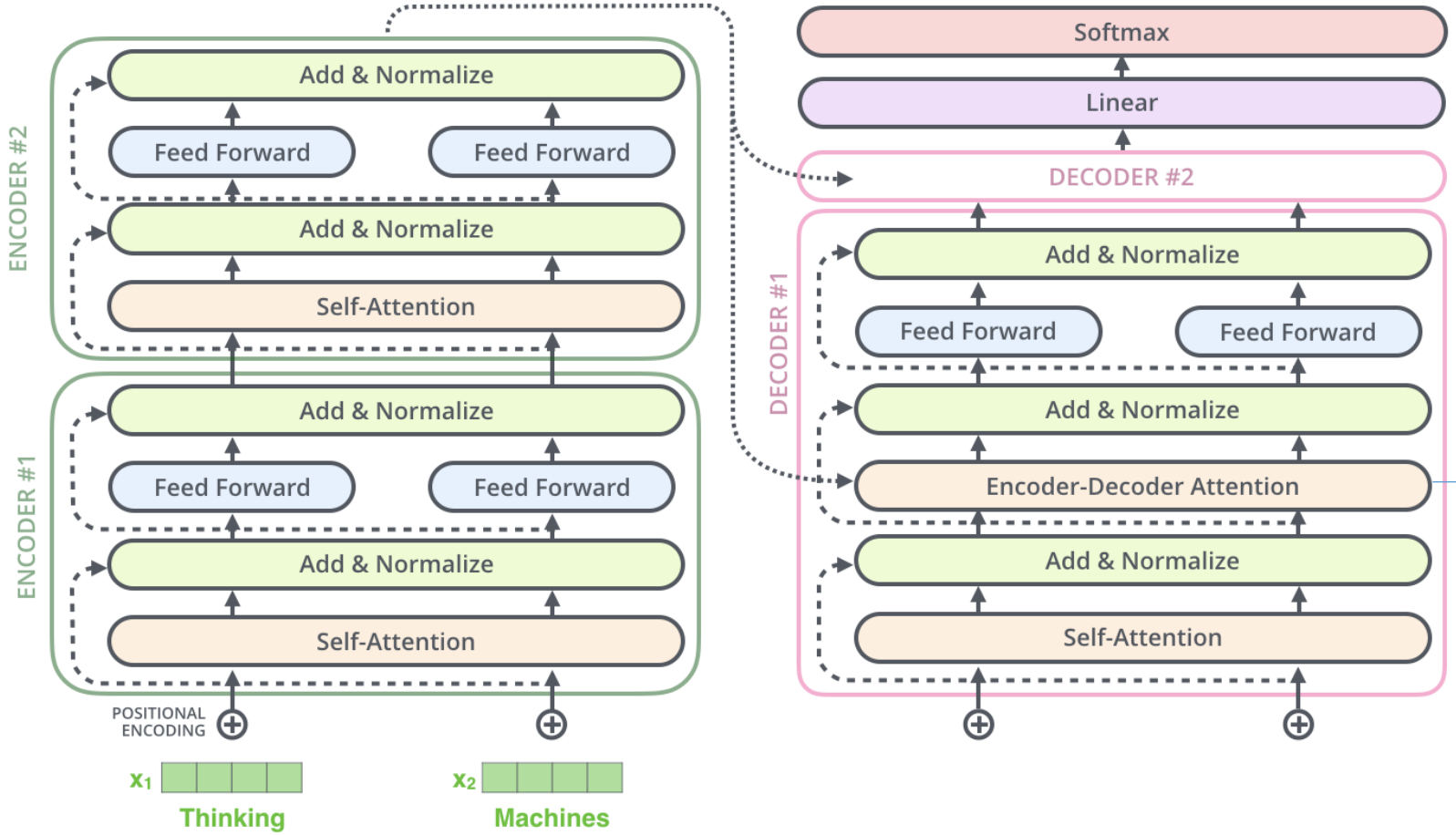
- A stack of $N = 6$ identical layers.
- Each layer has a multi-head self-attention layer and a simple position-wise fully connected feed-forward network.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- The linear transformations are the same across different positions, they use different parameters from layer to layer.
- Each sub-layer adopts a residual connection and a layer normalization.

Transformer Encoder

The outputs of encoder go to the sub-layers of the decoder as well. If we're to think of a Transformer of two stacked encoders and decoders, it would look something like this:

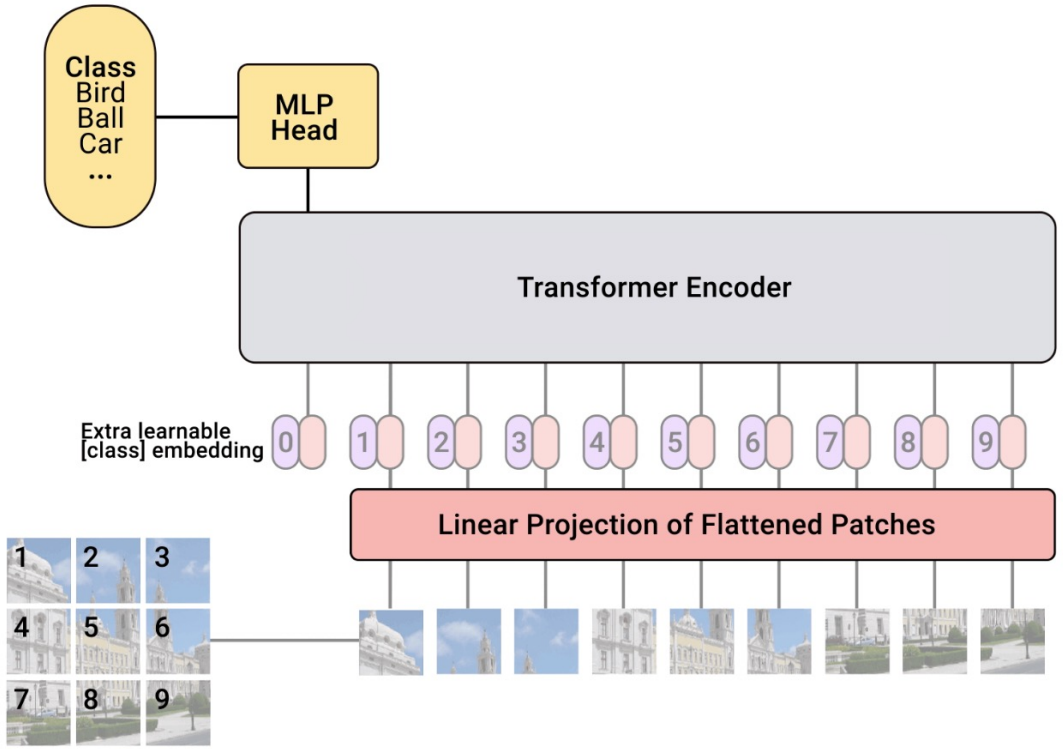


The "Encoder-Decoder Attention" layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

Vision Transformer (ViT)

- Do not have decoder

- Reshape the image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x} \in \mathbb{R}^{N \times (P^2 \cdot C)}$
  - (H, W) is the resolution of the original image
  - C is the number of channels
  - (P, P) is the resolution of each image patch
  - $N = HW/P^2$ is the resulting number of patches



Alexey Dosovitskiy et al., An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, ICLR 2021

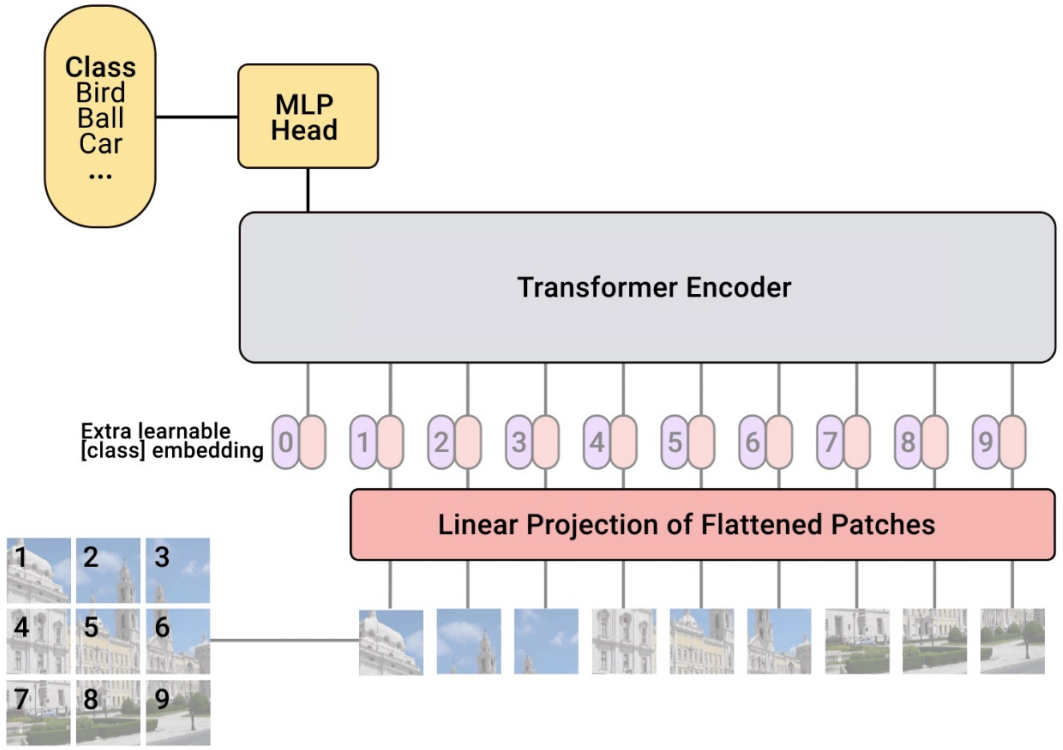
Vision Transformer (ViT)

Prepend a learnable embedding ($\mathbf{z}_0^0 = \mathbf{x}_{\text{class}}$) to the sequence of embedded patches

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}};\ \mathbf{x}_p^1\mathbf{E};\ \mathbf{x}_p^2\mathbf{E};\ \cdots\ ;\ \mathbf{x}_p^N\mathbf{E}] + \mathbf{E}_{pos}$$

Patch embedding - Linearly embed each of them to D dimension with a trainable linear projection $\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}$

Add learnable position embeddings $\mathbf{E}_{pos} \in \mathbb{R}^{(N+1) \times D}$ to retain positional information

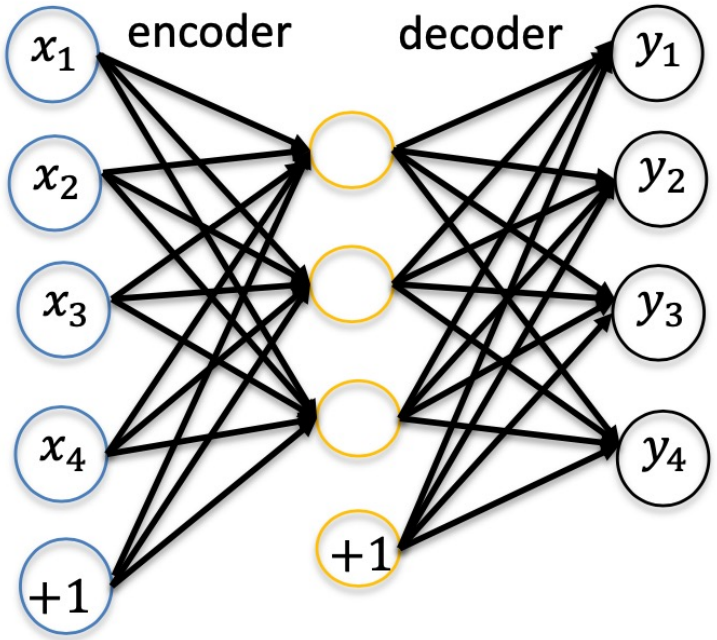


Alexey Dosovitskiy et al., An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, ICLR 2021

Outline – Autoencoders

- Supervised vs unsupervised learning
- Autoencoders
  - Denoising autoencoders
  - Undercomplete and overcomplete autoencoders
  - Sparse autoencoders
  - Other encoders

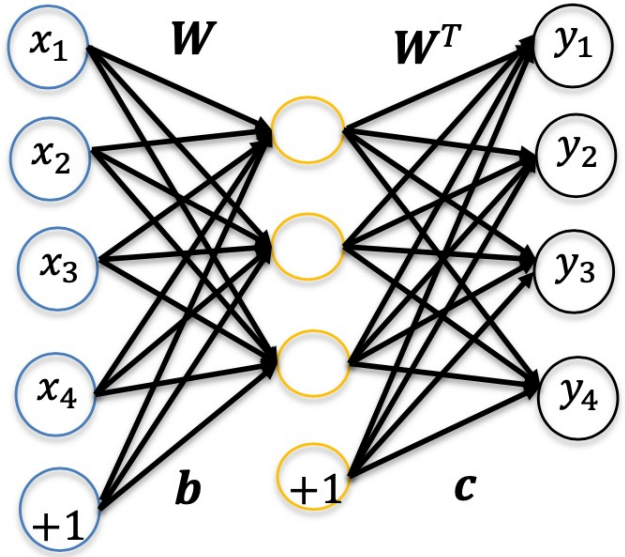
Autoencoders



An autoencoder is a neural network that is trained to attempt to copy its input to its output. It's hidden layer describes a *code* that represents the input.

The network consists of two parts: an encoder and a decoder.

Autoencoders



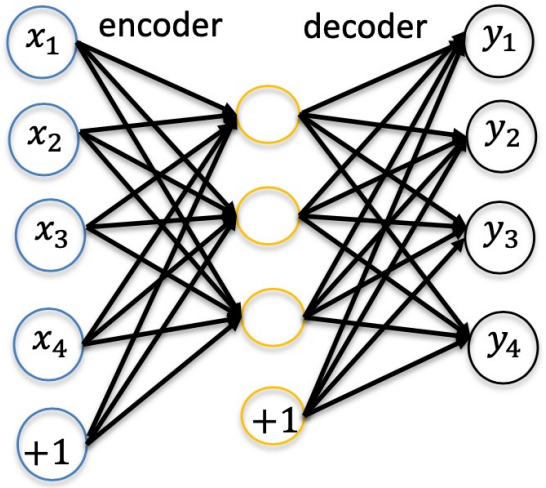
Reverse mapping from the hidden layer to the output can be optionally constrained to be the same as the input to hidden-layer mapping (tied weights). That is, if encoder weight matrix is $\boldsymbol{W}$, the decoder weigh matrix is $\boldsymbol{W^T}$.

Hidden layer and output layer activation can be then written as

$$\boldsymbol{h} = f(\boldsymbol{W^T x} + \boldsymbol{b})$$
$$\boldsymbol{y} = f(\boldsymbol{Wh} + \boldsymbol{c})$$

$\boldsymbol{f}$ is usually a sigmoid.

Denoising Autoencoders (DAE)



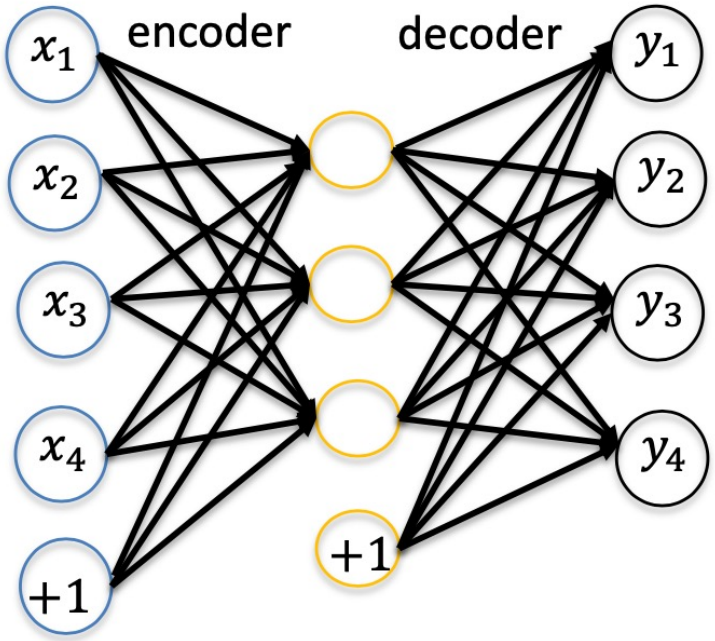
A *denoising autoencoder* (DAE) receives corrupted data points as inputs.

It is trained to predict the original uncorrupted data points as its output.

The idea of DAE is that in order to force the hidden layer to discover more robust features and prevent it from simply learning the identity. We train the autoencoder to reconstruct the input from a corrupted version of it.

In other words, DAE attempts to encode the input (preserve the information about input) and attempts to undo the effect of corruption process applied to the input of the autoencoder.

Autoencoders



Input dimension n and hidden dimension M :

If $M < n$, *undercomplete* autoencoders

If $M > n$, *overcomplete* autoencoders

Sparse Autoencoders (SAE)

A sparse autoencoder (SAE) is simply an autoencoder whose training criterion involves the sparsity penalty $\Omega_{sparsity}$ at the hidden layer:

$$J_1 = J + \beta\, \Omega_{sparsity}(h)$$

The sparsity penalty term makes the features (weights) learnt by the hidden-layer to be sparse.

With the sparsity constraint, one would constraint the neurons at the hidden layers to be inactive for most of the time.

We say that the neuron is "active" when its output is close to 1 and the neuron is "inactive" when its output is close to 0.

Sparsity constraint

To achieve sparse activations at the hidden-layer, the Kullback-Leibler (KL) divergence is used as the sparsity constraint:

$$D(\boldsymbol{h}) = \sum_{j=1}^{M} \rho \log \frac{\rho}{\rho_j} + (1-\rho) \log \frac{1-\rho}{1-\rho_j}$$

where M is the number of hidden neurons and ρ is the sparsity parameter.

KL divergence measures the deviation of the distribution $\{\rho_j\}$ of activations at the hidden-layer from the uniform distribution of ρ.

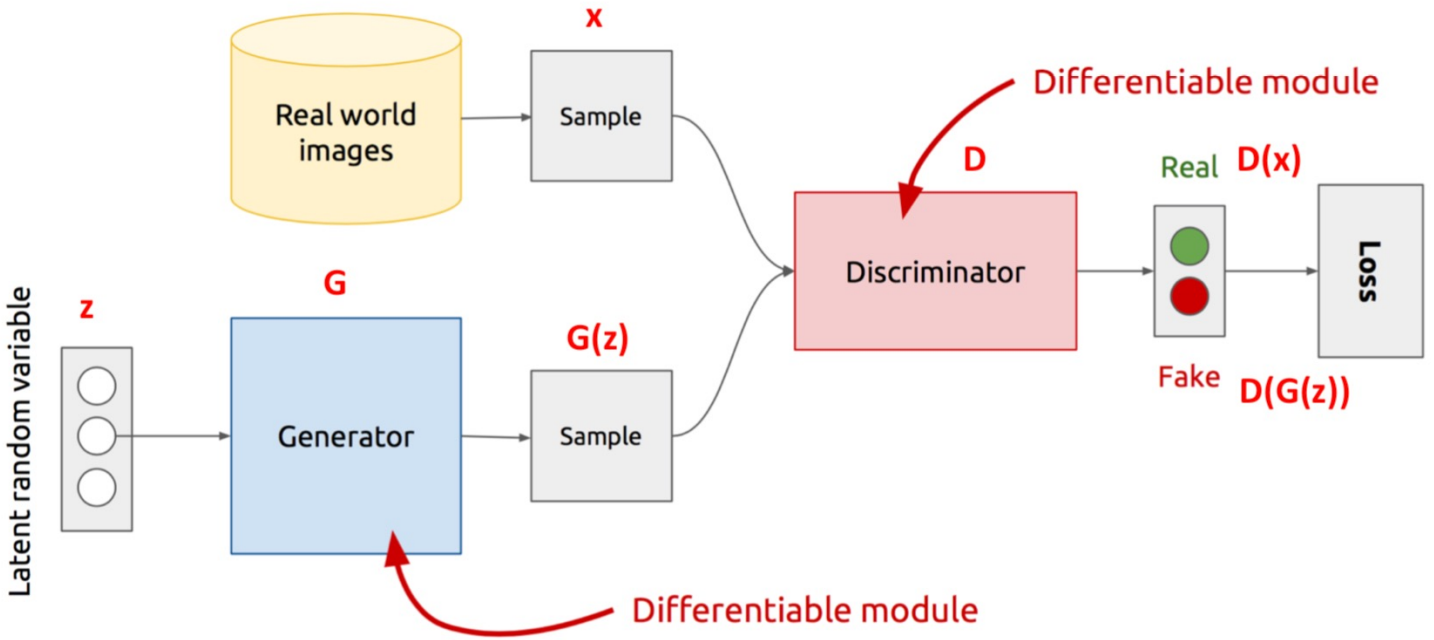
The KL divergence is minimum when $\rho_j = \rho$ for all j.
That is, when the average activations are uniform and equal to very low value ρ.

Outline – GAN

- GAN Basics
- GAN Training
- DCGAN
- Mode Collapse

Generative adversarial networks (GAN)

- Generative model G:
  - Captures data distribution
  - Fool $D(G(z))$
  - Generate an image $G(z)$ such that $D(G(z))$ is wrong (i.e. $D(G(z)) = 1$)
- Discriminative model D:
  - Distinguishes between real and fake samples
  - $D(x) = 1$ when x is a real image, and otherwise $D(x) = 0$



z is some random noise (Gaussian/Uniform).

z can be thought as the latent representation of the data.

Generative adversarial networks (GAN)

- D and G play the following two-player minimax game with value function $V(G, D)$

Generator: input = random vector z
Output: $G(z)$ as synthetic image

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Discriminator: input = generated/real images
Output: $D(x) = 1$ if real image

Training procedure

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient: maximize

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

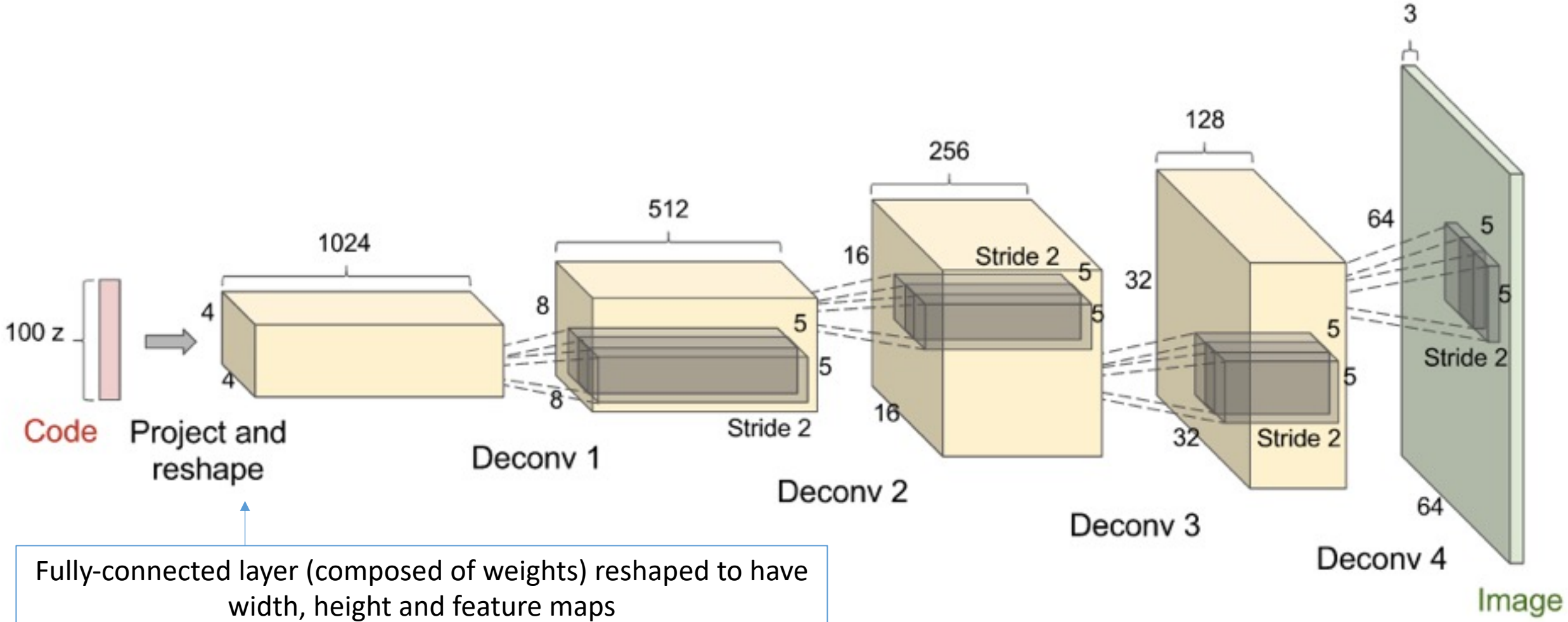
  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient: minimize

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Deep Convolutional GAN (DCGAN)

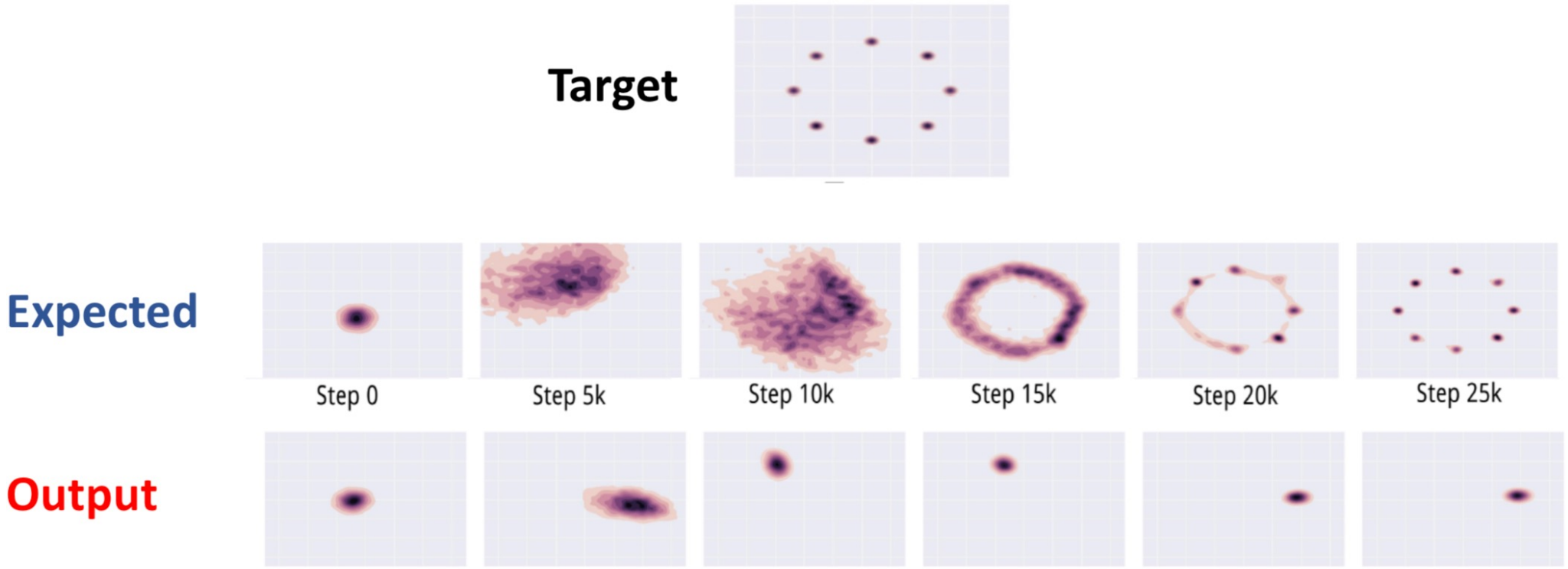


Fully-connected layer (composed of weights) reshaped to have width, height and feature maps

We will go through the code in the tutorial

Mode Collapse

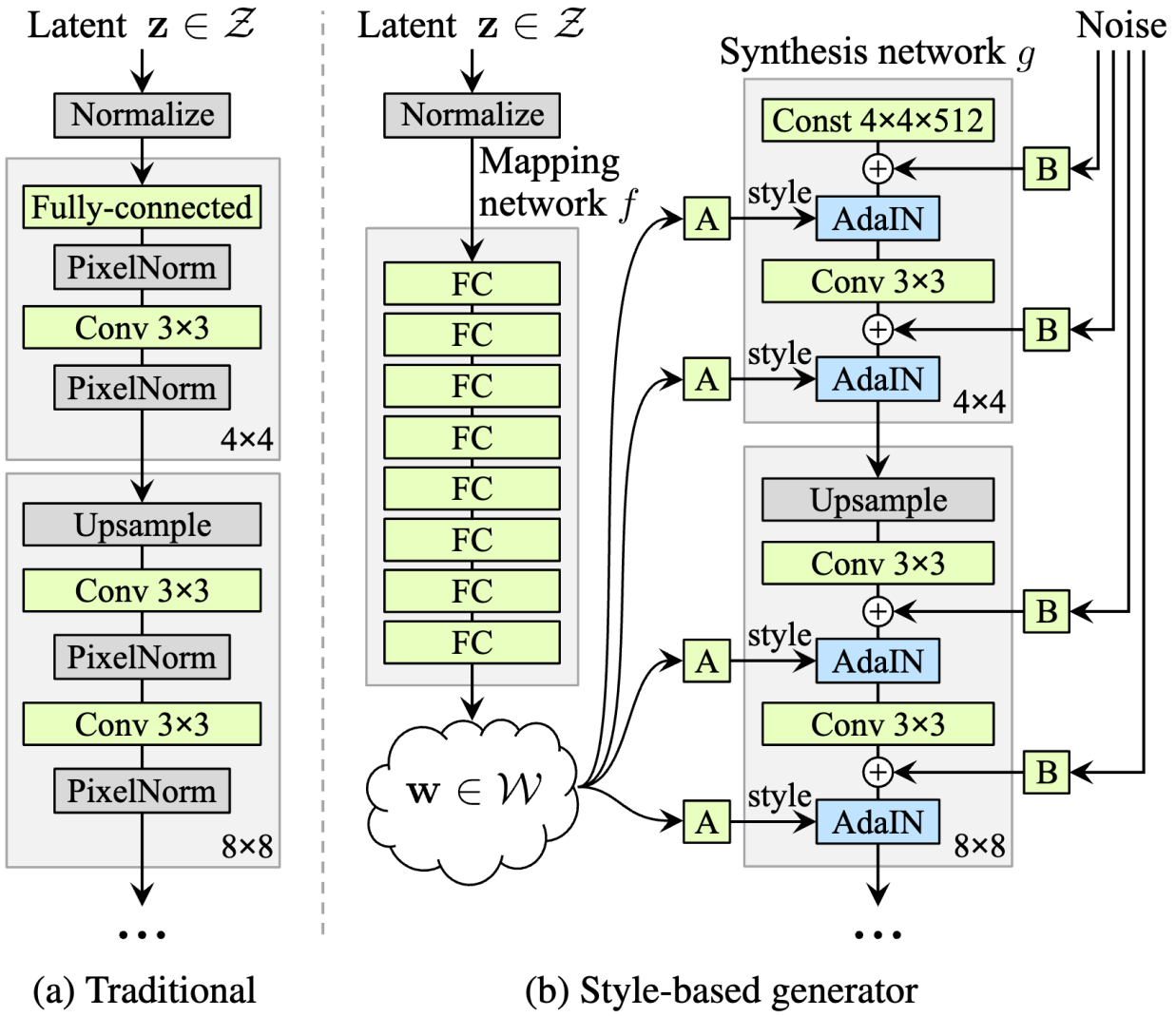
Generator fails to output diverse samples



Metz, Luke, et al. "Unrolled Generative Adversarial Networks." arXiv preprint arXiv:1611.02163 (2016)

Applications of GANs
(not included in the final exam)

StyleGAN



(a) Traditional (b) Style-based generator

A = learned affine transformation block for AdaIN (predicts y)

Adaptive Instance Normalization (very effective in controlling styles)

$$\mathrm{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i},$$

B = learned per-channel scaling factor for noise input.

Karras et al, A Style-Based Generator Architecture for Generative Adversarial Networks, CVPR2019

Coarse styles
($4^2 – 8^2$)

https://www.youtube.com/watch?v=kSLJriaOumA

BigGAN

Brock et al, Large Scale GAN Training for High Fidelity Natural Image Synthesis, ICLR 2019

BigGAN

Batch	Ch.	Param (M)	Shared	Hier.	Ortho.	Itr $\times 10^3$	FID	IS
256	64	81.5	SA-GAN Baseline			1000	18.65	52.52
512	64	81.5	✗	✗	✗	1000	15.30	58.77($\pm$1.18)
1024	64	81.5	✗	✗	✗	1000	14.88	63.03($\pm$1.42)
2048	64	81.5	✗	✗	✗	732	12.39	76.85($\pm$3.83)
2048	96	173.5	✗	✗	✗	295($\pm$18)	9.54($\pm$0.62)	92.98($\pm$4.27)
2048	96	160.6	✓	✗	✗	185($\pm$11)	9.18($\pm$0.13)	94.94($\pm$1.32)
2048	96	158.3	✓	✓	✗	152($\pm$7)	8.73($\pm$0.45)	98.76($\pm$2.84)
2048	96	158.3	✓	✓	✓	165($\pm$13)	8.51($\pm$0.32)	99.31($\pm$2.10)
2048	64	71.3	✓	✓	✓	371($\pm$7)	10.48($\pm$0.10)	86.90($\pm$0.61)

Table 1: Fréchet Inception Distance (FID, lower is better) and Inception Score (IS, higher is better) for ablations of our proposed modifications. *Batch* is batch size, *Param* is total number of parameters, *Ch.* is the channel multiplier representing the number of units in each layer, *Shared* is using shared embeddings, *Hier.* is using a hierarchical latent space, *Ortho.* is Orthogonal Regularization, and *Itr* either indicates that the setting is stable to 10^6 iterations, or that it collapses at the given iteration. Other than rows 1-4, results are computed across 8 different random initializations.

BigGAN – Interpolations between c, z pairs

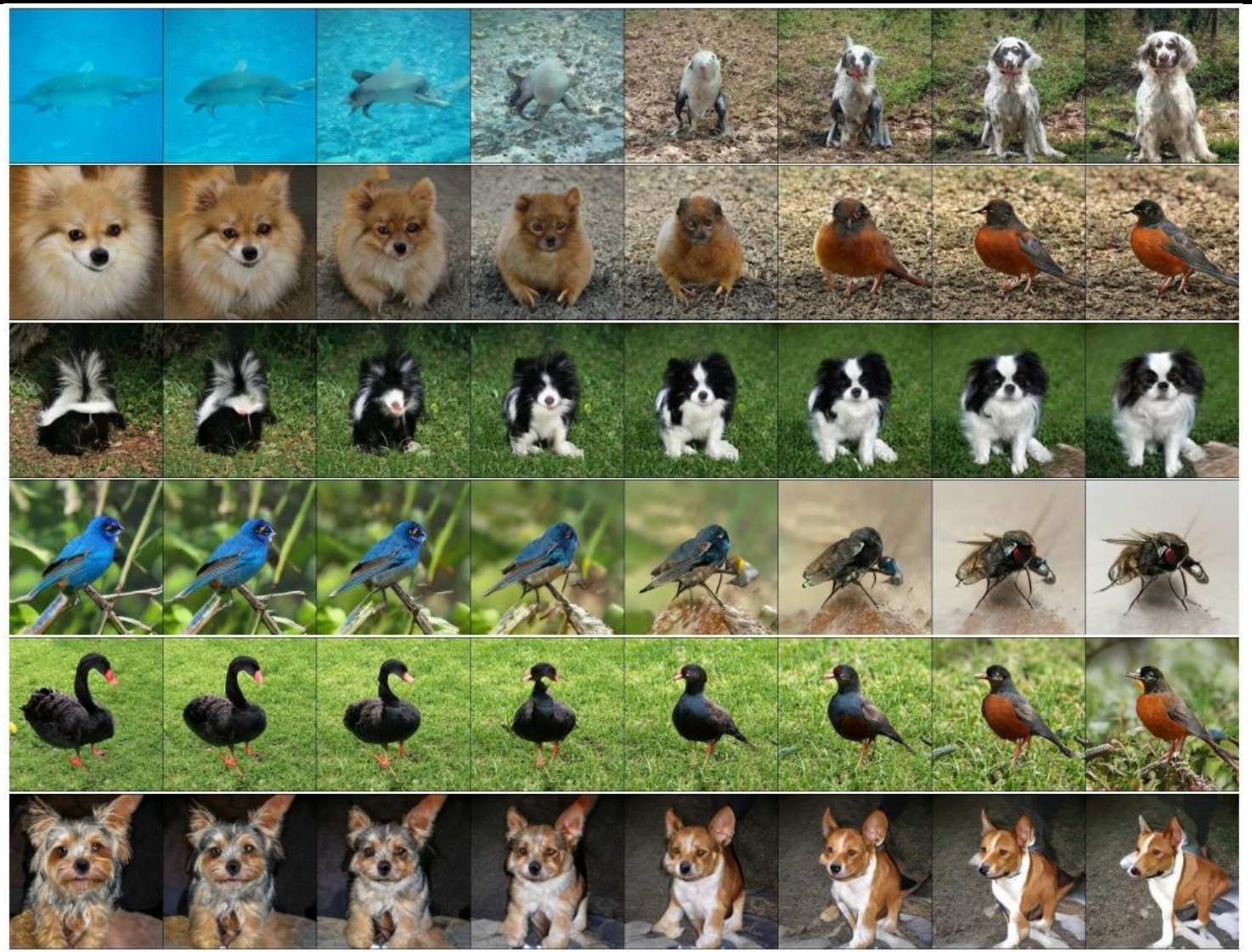


Figure 8: Interpolations between z, c pairs.

GAN as a Prior for Image Restoration and Manipulation

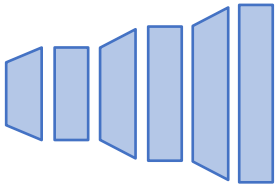
Xingang Pan, et al., Exploiting Deep Generative Prior for Versatile Image Restoration and Manipulation, ECCV 2020

Motivation

Our goal: exploit generic image prior of pretrained GAN

$$\mathbf{z} \sim \mathcal{N}(0, I)$$

Latent space



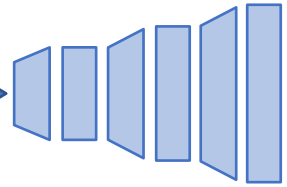
Generator

gray

hole

low resolution

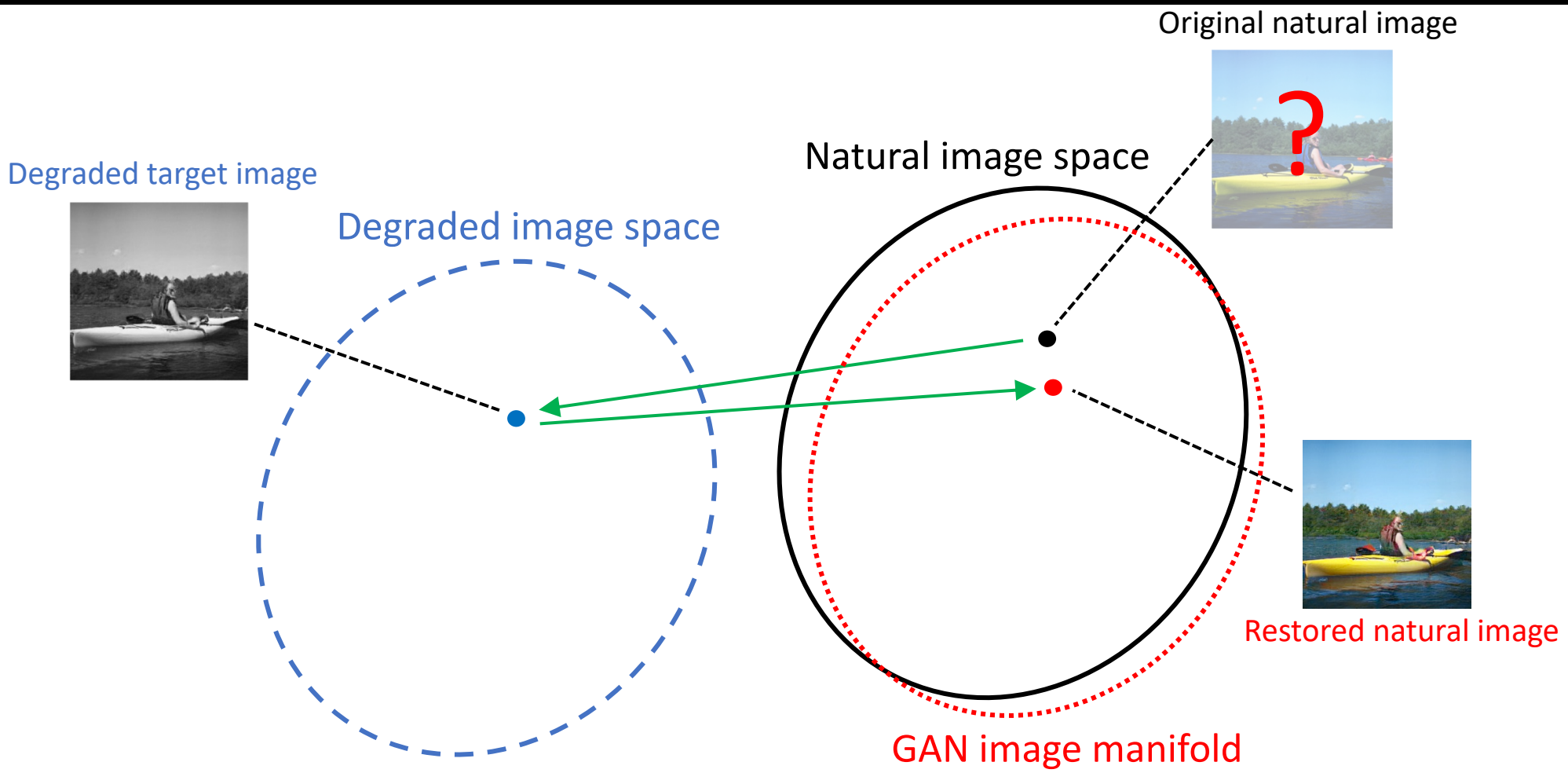
degraded images



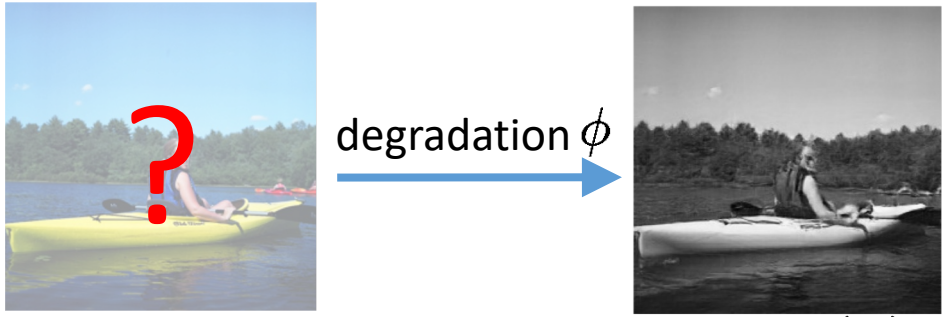
Generator

restored images

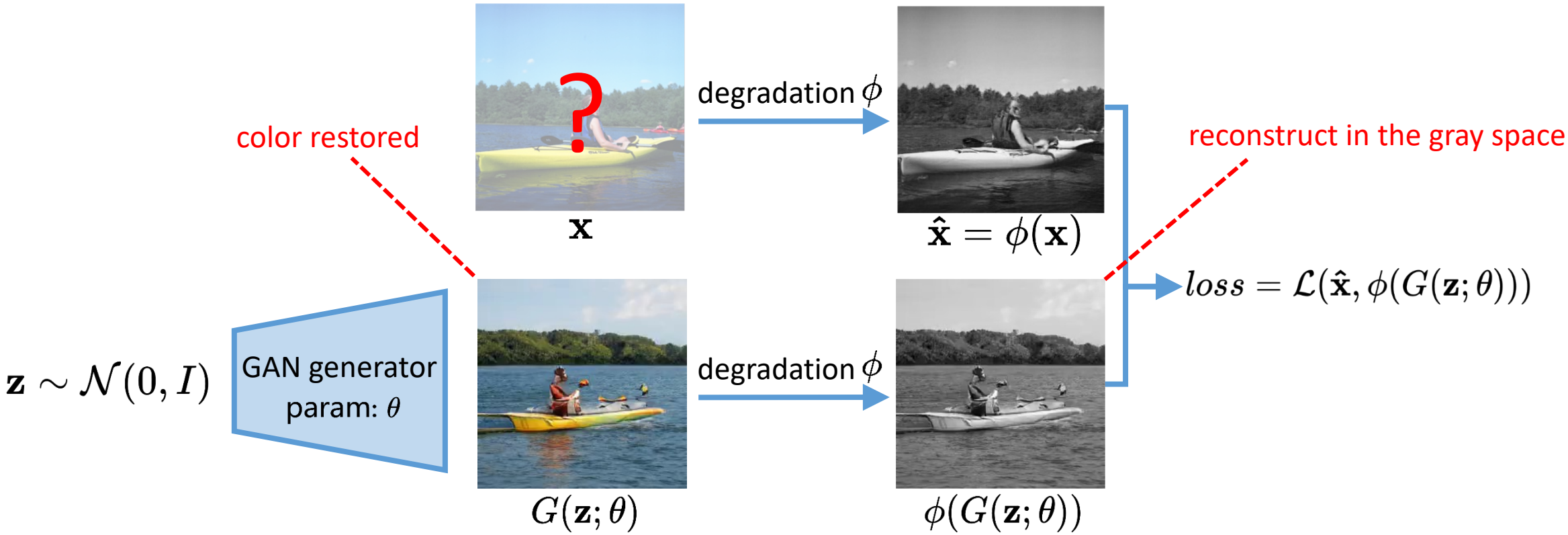
Deep Generative Prior



Deep Generative Prior



Deep Generative Prior



Deep Generative Prior

Conventional GAN-Inversion:

Optimize

$\mathbf{z} \sim \mathcal{N}(0, I)$

GAN generator
param: θ

$\mathbf{x}$

degradation ϕ

$\hat{\mathbf{x}} = \phi(\mathbf{x})$

$G(\mathbf{z}; \theta)$

degradation ϕ

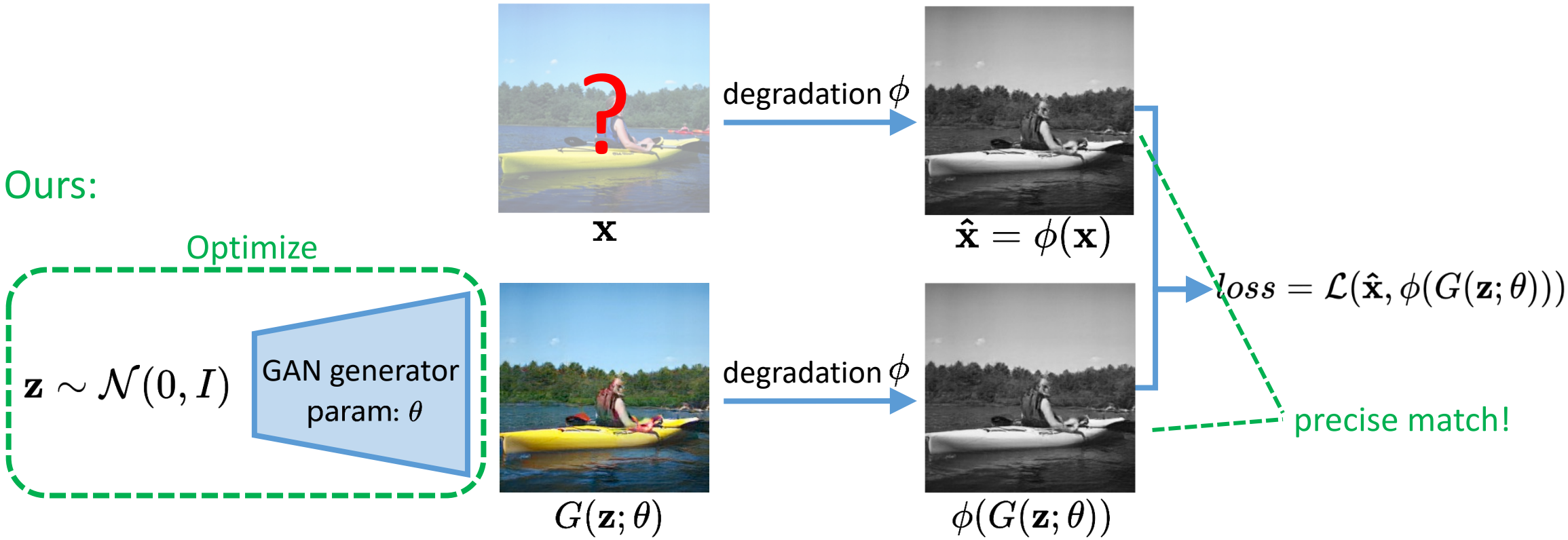
$\phi(G(\mathbf{z}; \theta))$

$loss = \mathcal{L}(\hat{\mathbf{x}}, \phi(G(\mathbf{z}; \theta)))$

mismatch

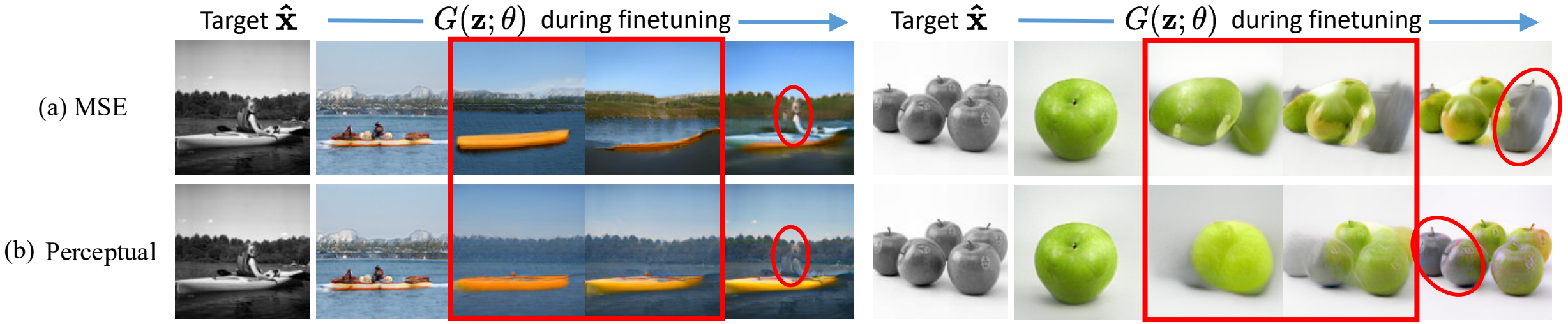
$$\mathbf{z}^* = argmin_{\mathbf{z} \in R^d} \mathcal{L}(\hat{\mathbf{x}}, \phi(G(\mathbf{z}; \theta)))$$

Deep Generative Prior



$$\theta^*, \mathbf{z}^* = argmin_{\theta,\mathbf{z}} \mathcal{L}(\hat{\mathbf{x}}, \phi(G(\mathbf{z};\theta))) \quad \text{(Relaxed GAN-inversion)}$$

Conventional Loss



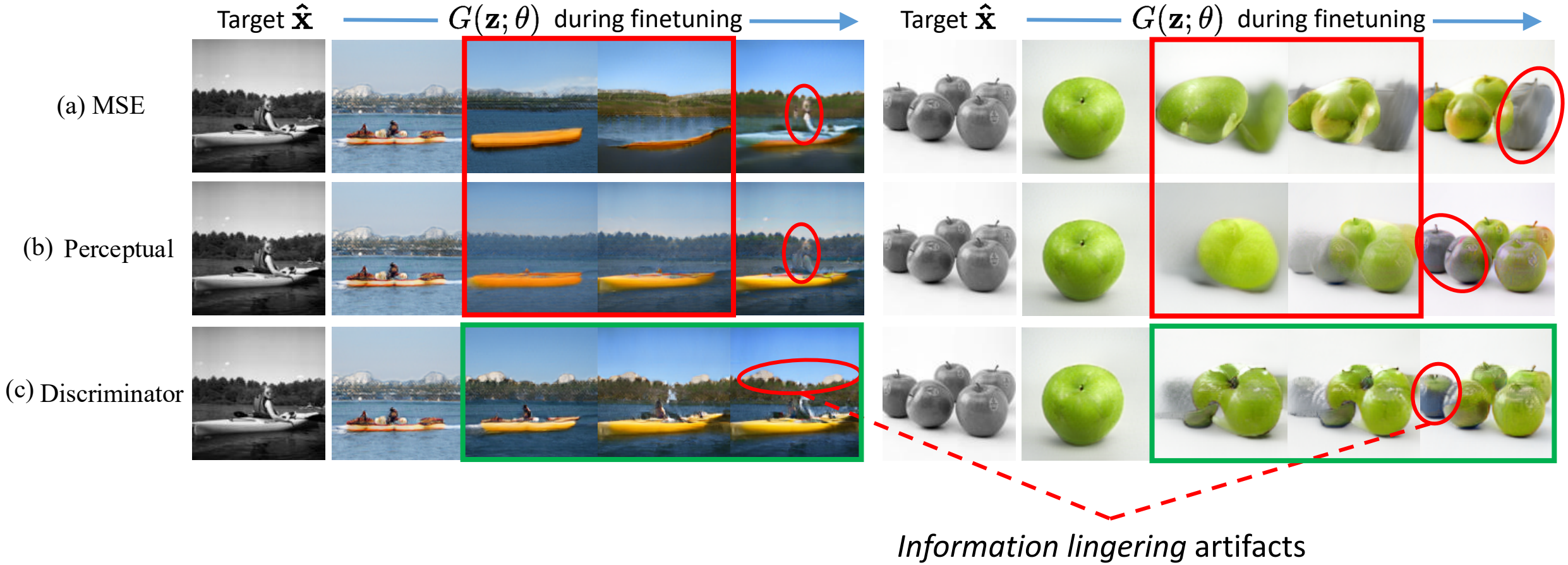
Discriminator Feature Matching Loss

L1 distance in the discriminator feature space:

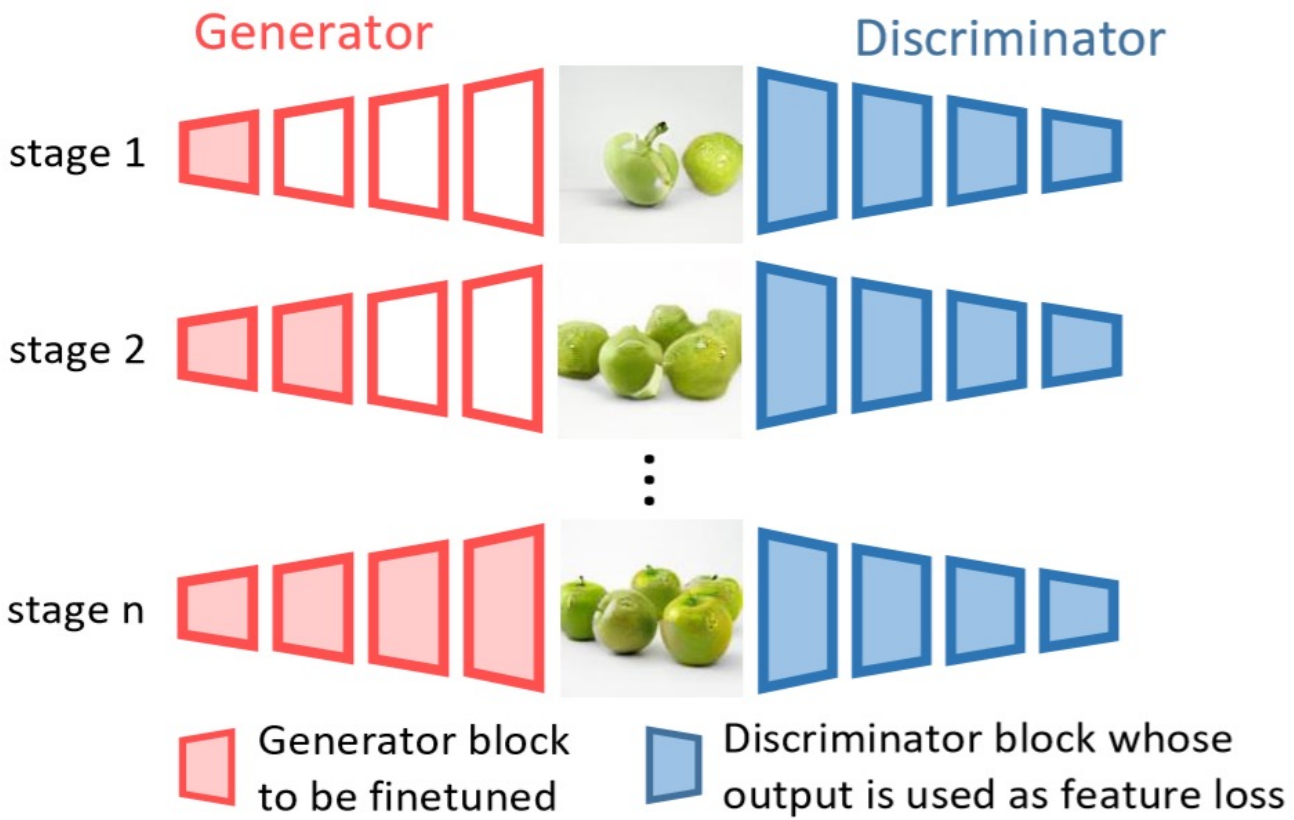
$$\mathcal{L}(\mathbf{x}_1, \mathbf{x}_2) = \sum_{i \in \mathcal{I}} \| D(\mathbf{x}_1, i), D(\mathbf{x}_2, i) \|_1$$

$D(\mathbf{x},\mathrm{i})$ returns the feature of $\mathbf{x}$ at the i'th block of the discriminator.

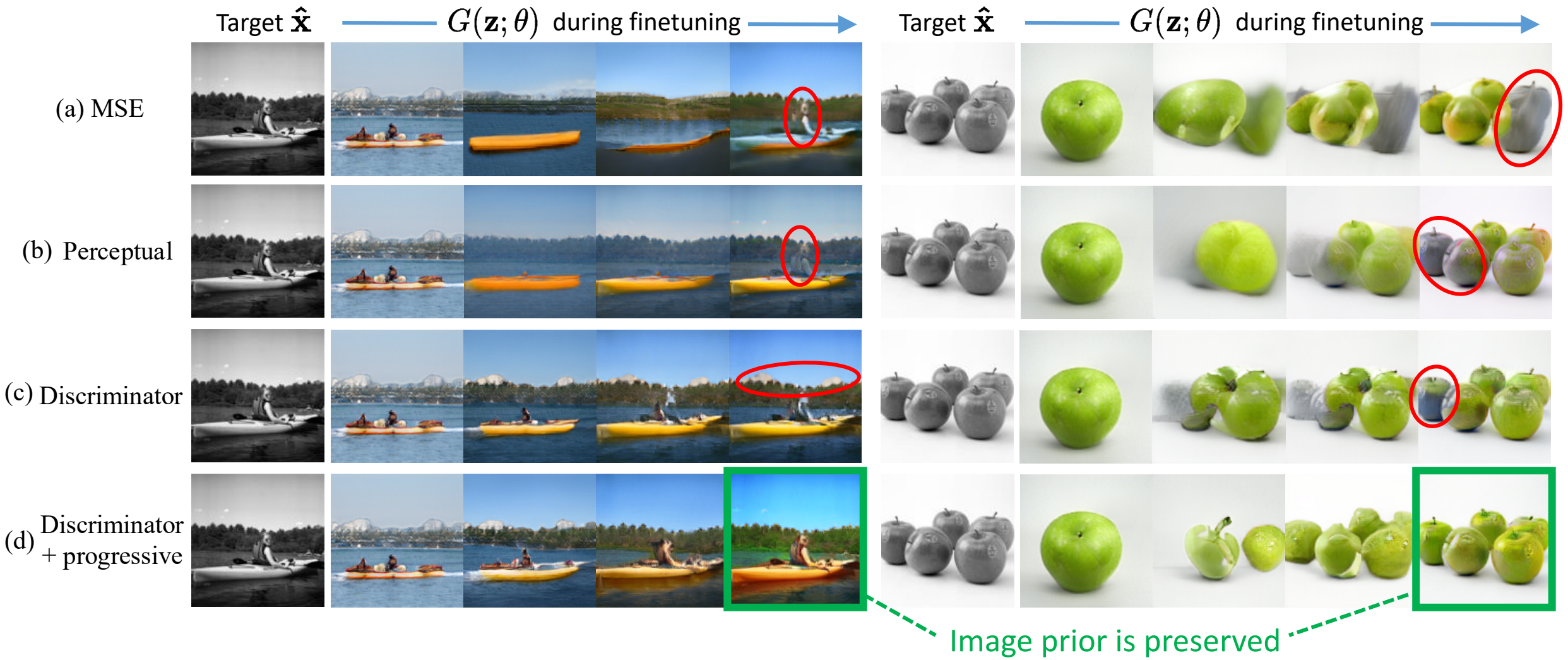
Discriminator Feature Matching Loss



Progressive Reconstruction



Comparison of Different Losses



Applications

Degradation transform $\phi(\mathbf{x})$ for different tasks:

Colorization: $\phi(\mathbf{x}) = 0.2989\mathbf{x}_r + 0.5870\mathbf{x}_g + 0.1140\mathbf{x}_b$

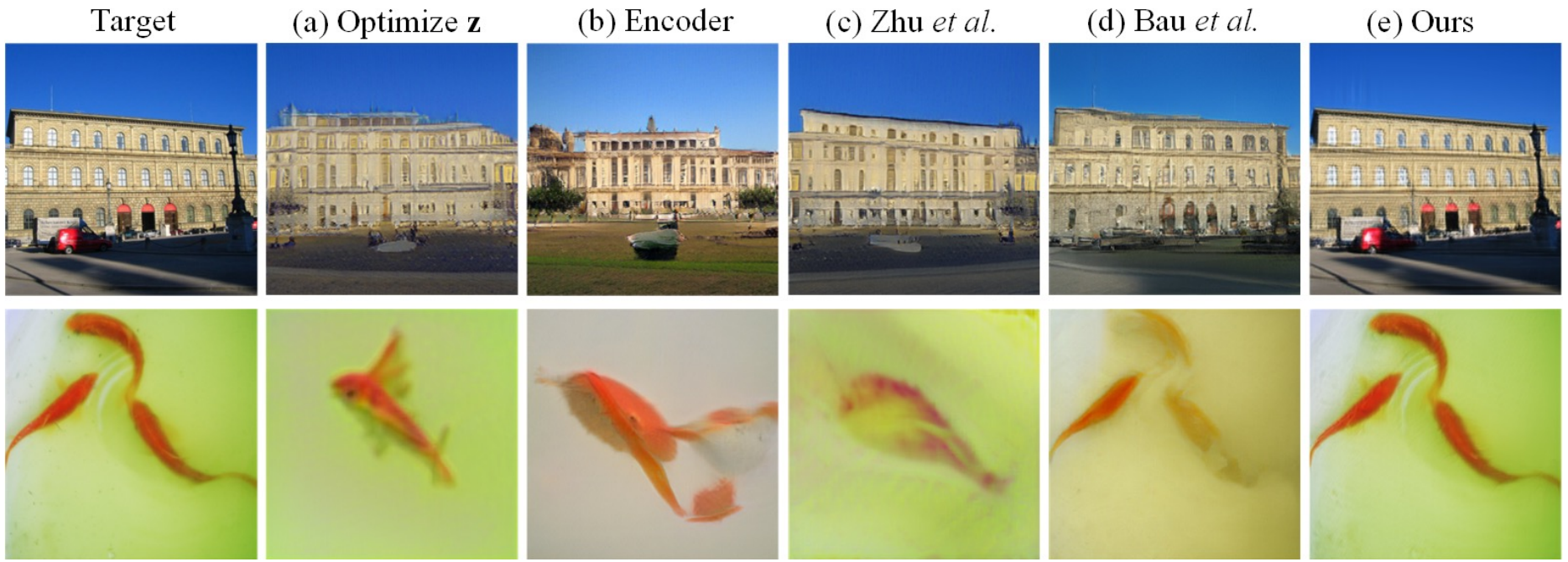
Inpainting: $\phi(\mathbf{x}) = \mathbf{x} \odot \mathbf{m}$ $\mathbf{m}$ is inpainting mask

Super-resolution: $\phi(\mathbf{x})$ is Lanczos downsampling operator

Adversarial defense: $\phi(\mathbf{x}) = \mathbf{x} + \Delta\mathbf{x}$ $\Delta\mathbf{x}$ is the adversarial perturbation

Model: BigGAN trained on ImageNet training set (Brock et al. ICLR2018)

Application: GAN Inversion

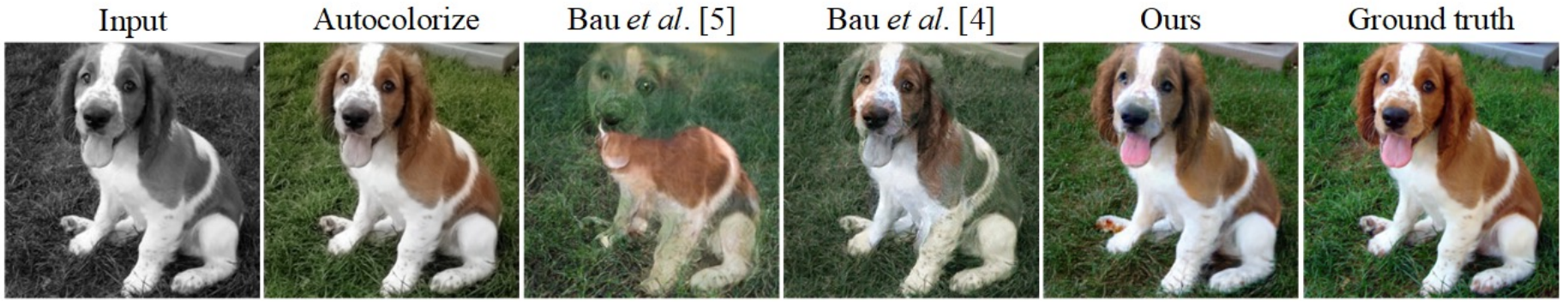


	(a)	(b)	(c)	(d)	Ours
PSNR↑	15.97	11.39	16.46	22.49	**32.89**
SSIM↑	46.84	32.08	47.78	73.17	**95.95**
MSE↓ (×e-3)	29.61	85.04	28.32	6.91	**1.26**

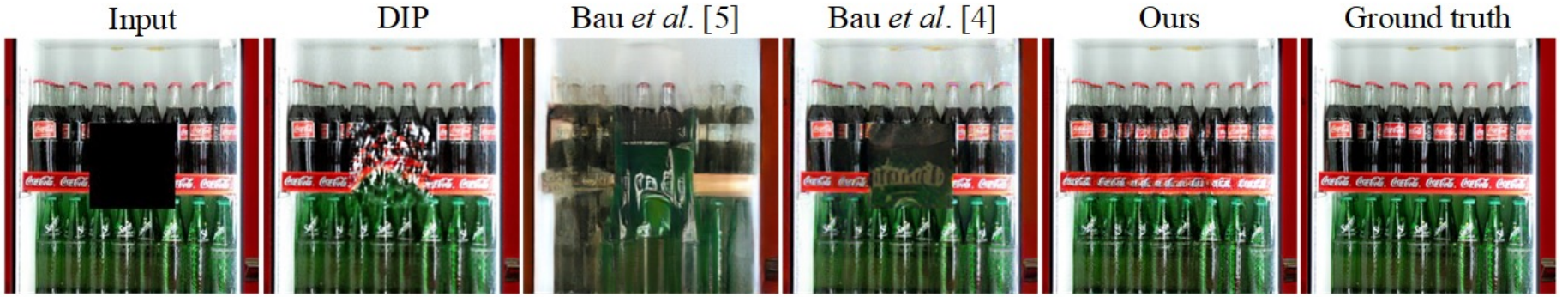
(a) Optimize z (Creswell et al. 2018)
(b) Encoder (Zhu et al. ECCV2016)
(c) Encoder + Optimize z (Zhu et al. ECCV2016)
(d) Encoder + Optimize z + Perturbation on features (Bau et al. ICCV2019)

Application: Image Restoration

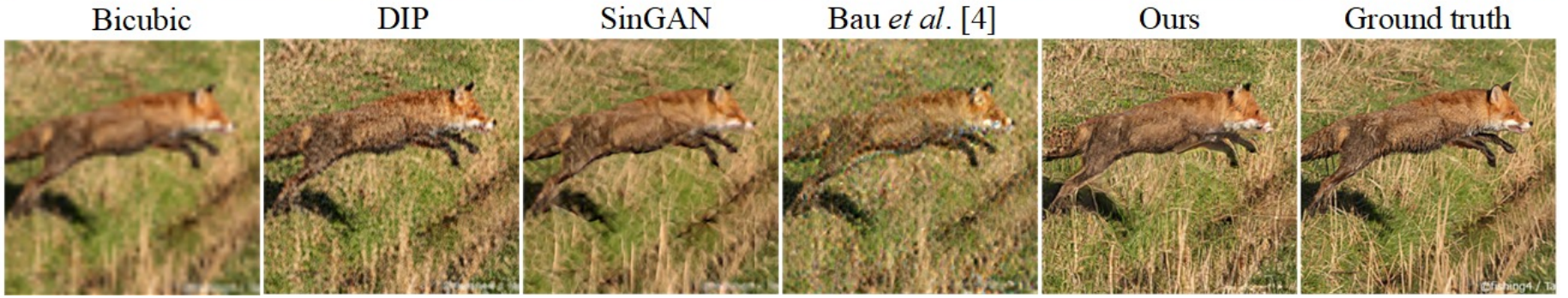
Colorization



Inpainting



Super-resolution



Demo: Image Restoration

Colorization

Inpainting

Super-resolution

Application: Image Restoration

Colorization

Inpainting

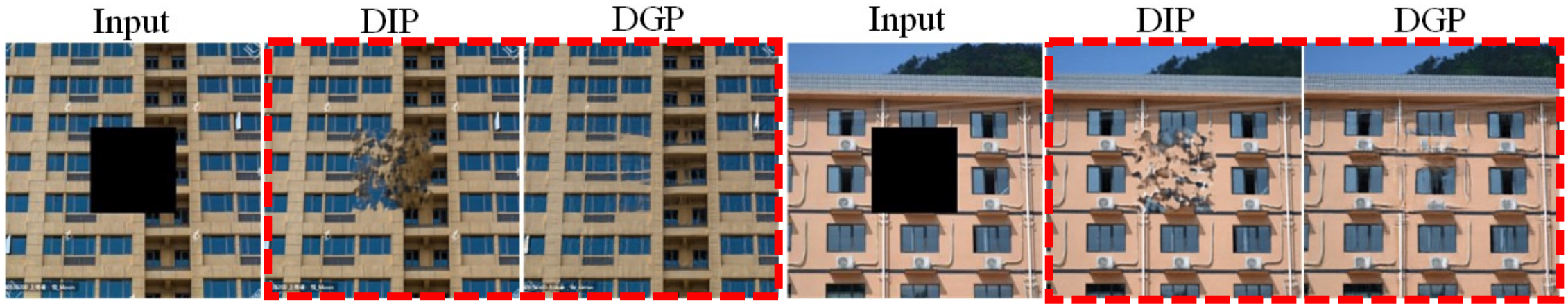
Super-Resolution

Generalization to non-ImageNet Images

(a) Raccoon

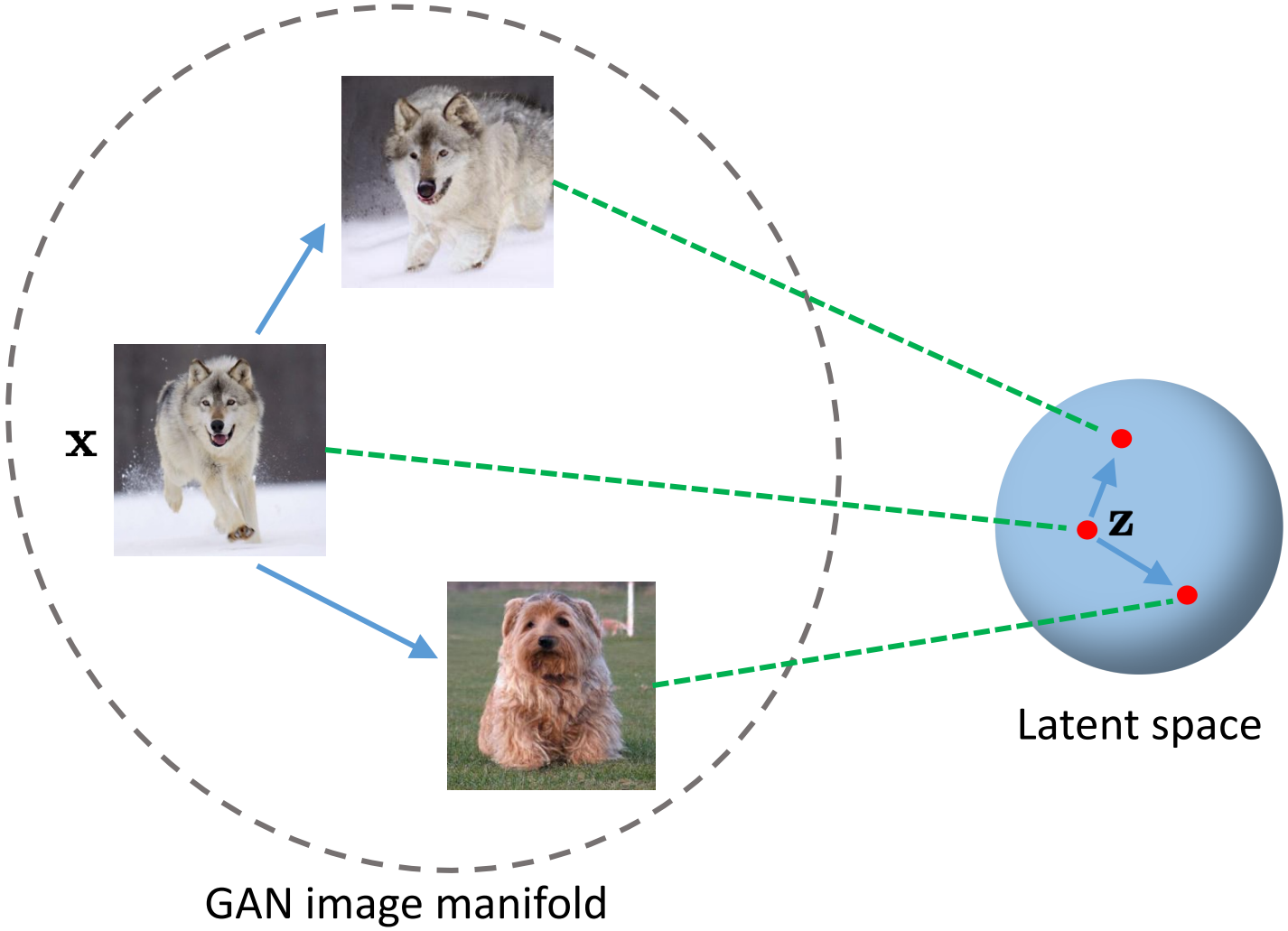
(b) Places

(c) No foreground

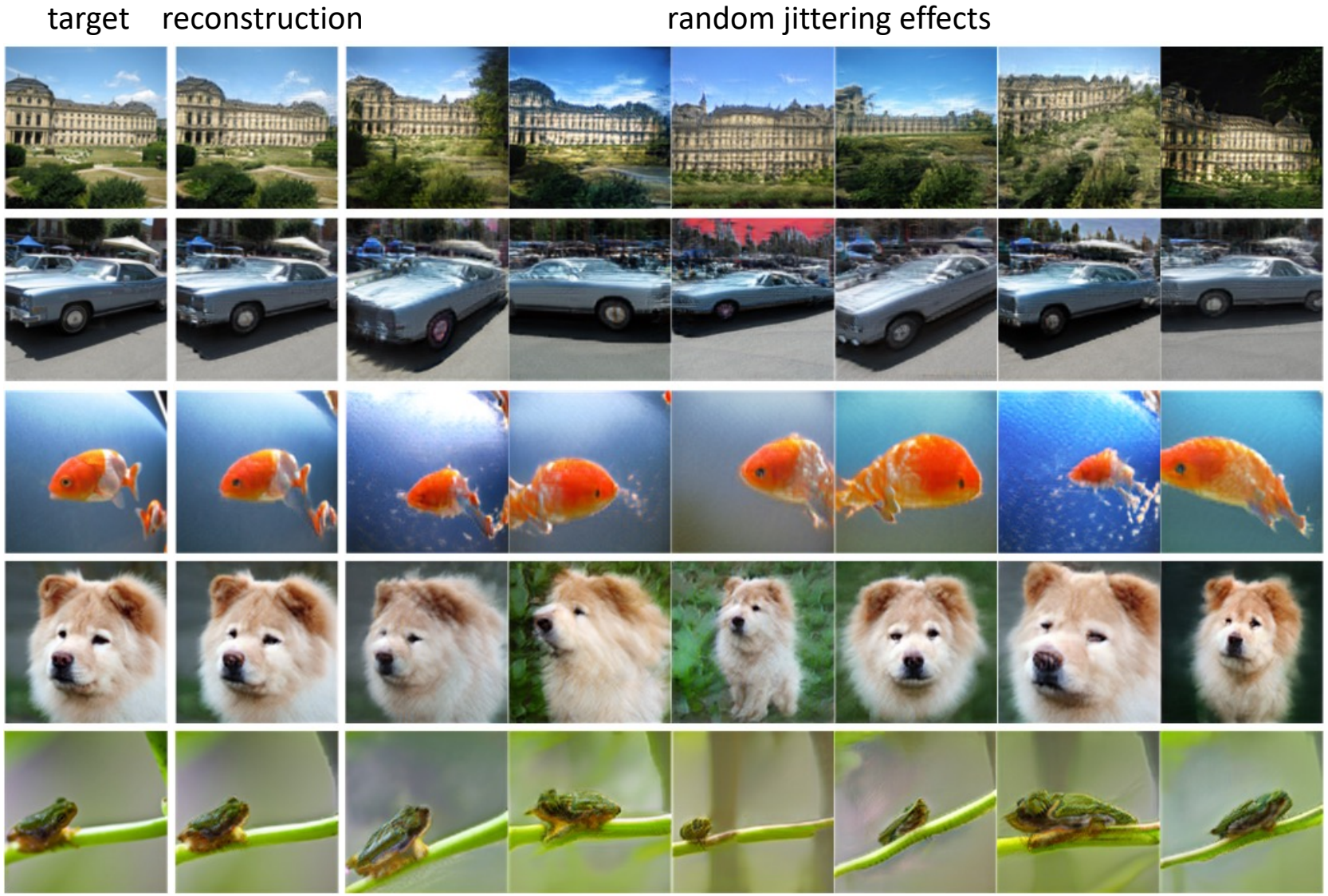


(d) Windows

Application: Image Manipulation



Application: Random Jittering



Application: Image Morphing

Target A

Generator
θ_A

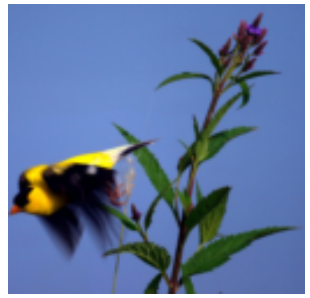
z_A

interpolate

interpolate

Generator
θ_B

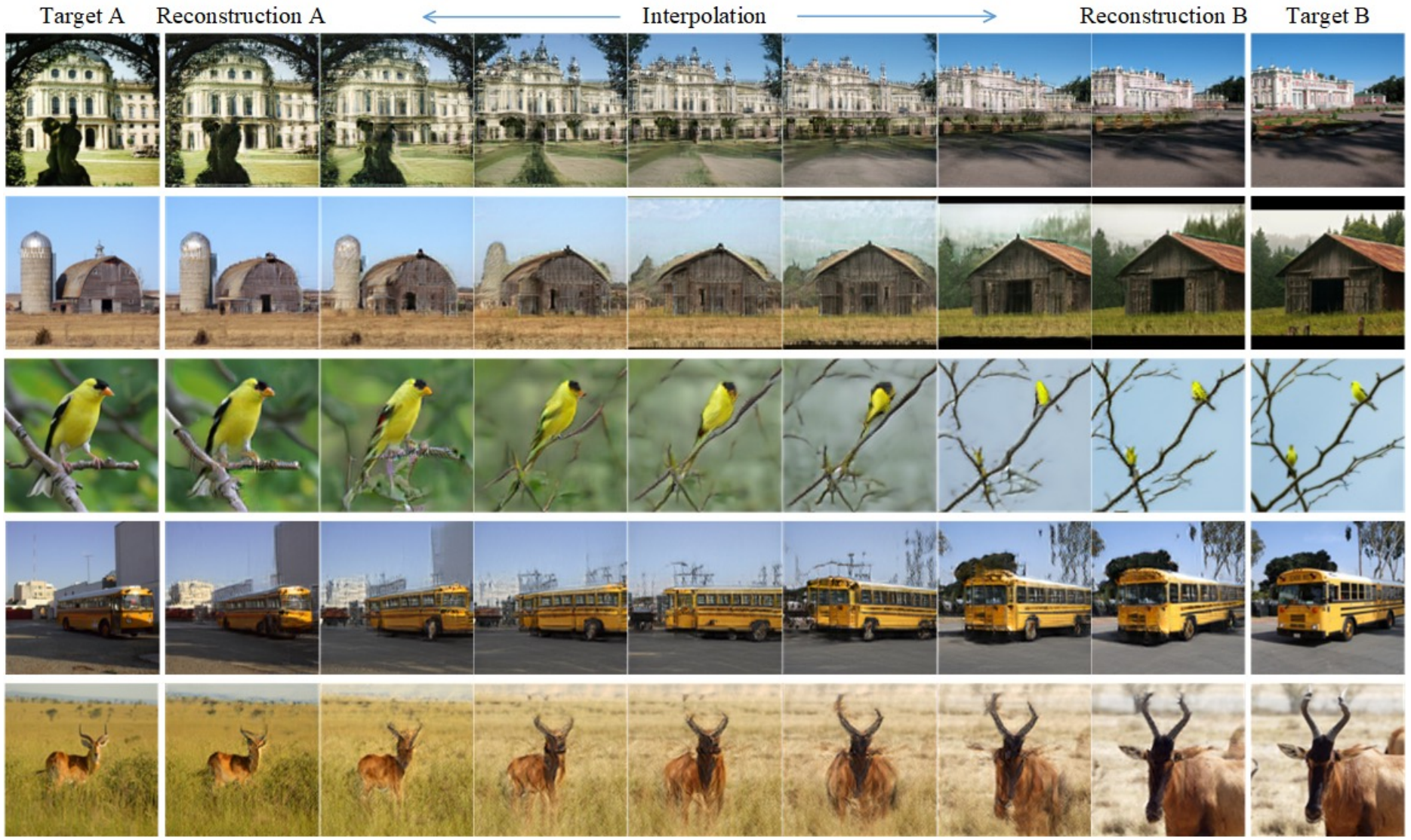
z_B



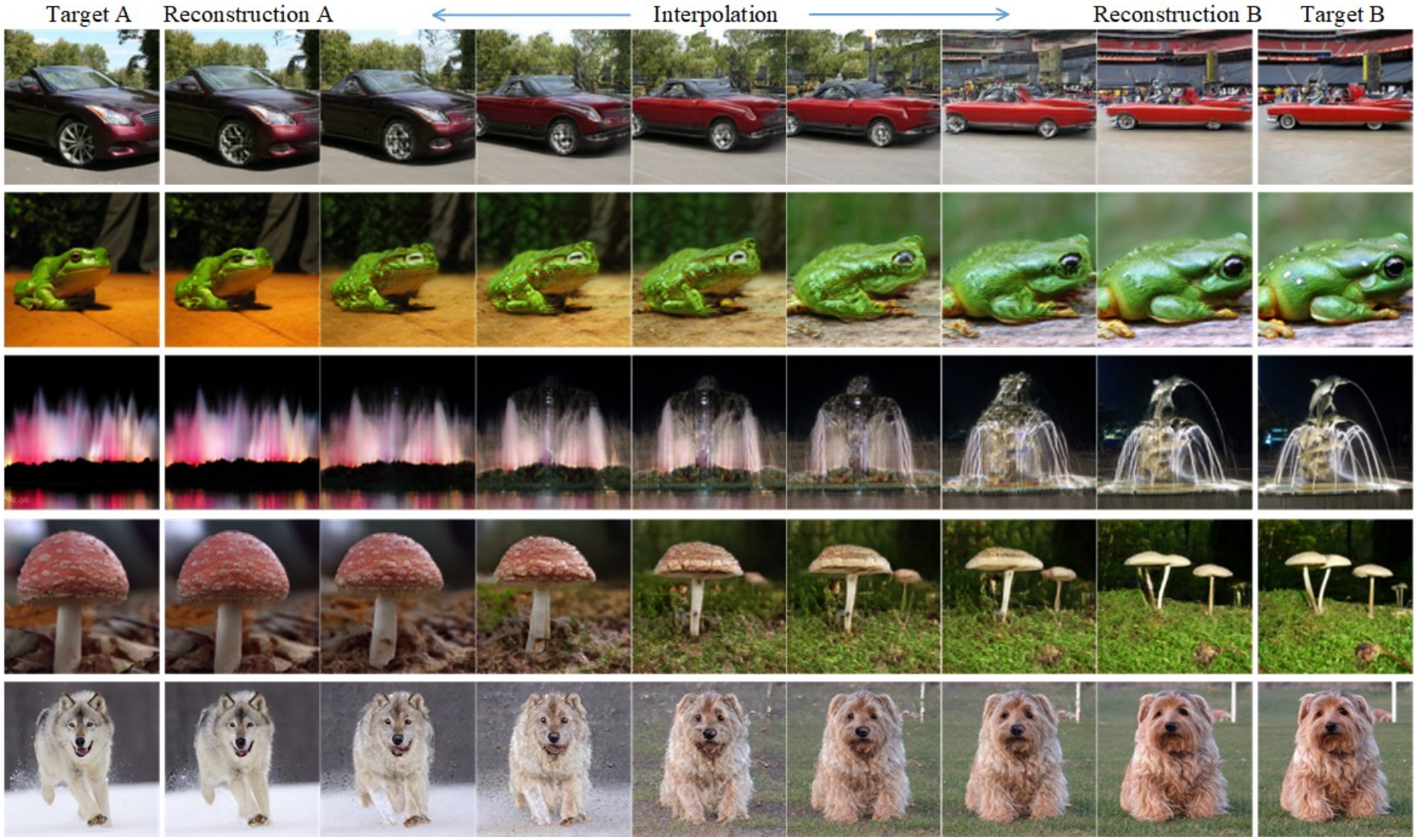
Target B

Application: Image Morphing

Application: Image Morphing



Application: Image Morphing



Application: Category Transfer

target reconstruct

transfer to other categories

Demo: Image Manipulation

target reconstruction

target reconstruction

target reconstruction

Random jittering

Image morphing

Category transfer

Exploiting 2D GAN Prior for 3D Generation

Xingang Pan, et al., Do 2D GANs Know 3D Shape? Unsupervised 3D Shape Reconstruction from 2D Image GANs. ICLR 2021

Do 2D GANs Model 3D Geometry?

Natural images are projections of 3D objects on a 2D image plane.

An ideal 2D image manifold (e.g., GAN) should capture 3D geometric properties.

The following example shows that there is a direction in the GAN image manifold that corresponds to viewpoint variation.

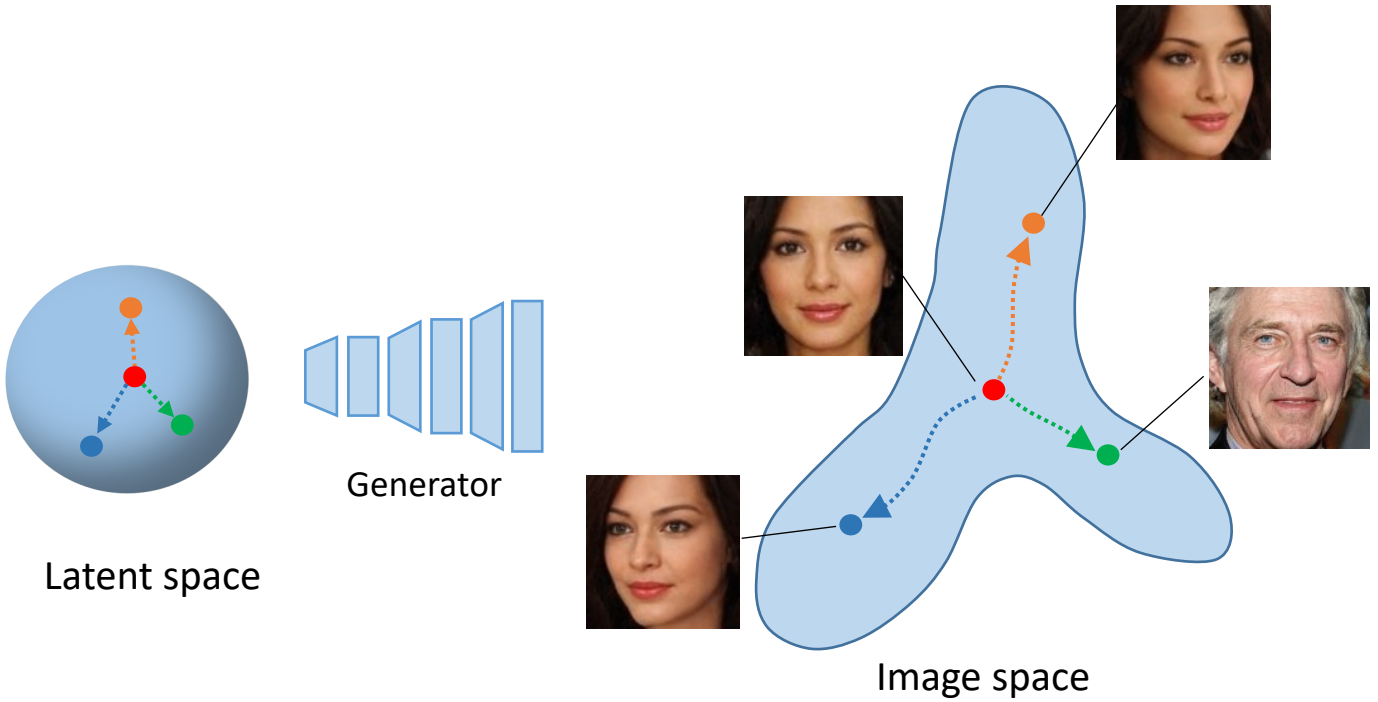
Can we Make Use of such Variations?

Can we make use of such variations for 3D reconstruction?

If we have multiple viewpoint and lighting variations of the same instance, we can infer its 3D structure.

Let's create these variations by exploiting the image manifold captured by 2D GANs!

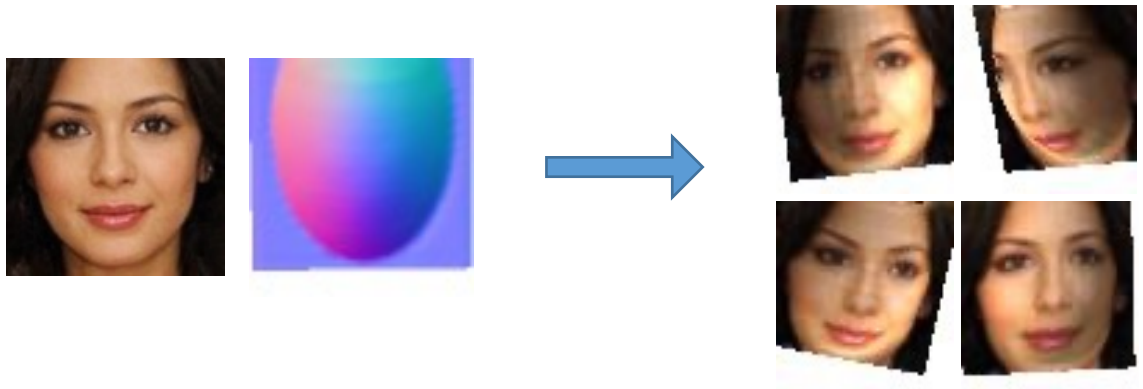
Challenge



It is non-trivial to find well-disentangled latent directions that control *viewpoint* and *lighting* variations in an unsupervised manner.

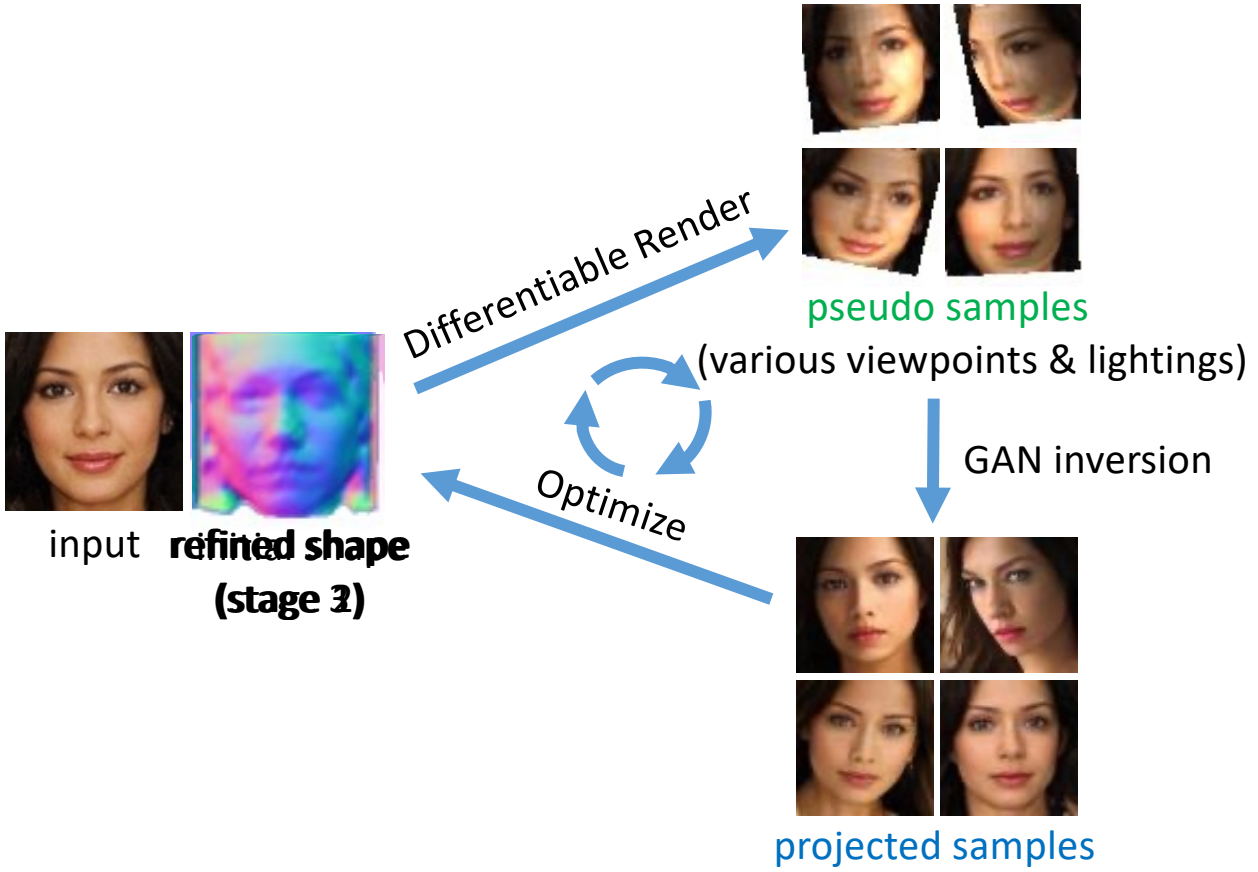
Our Solution

For many objects such as faces and cars, a convex shape prior like ellipsoid could provide a hint on the change of their viewpoints and lighting conditions.



GAN2Shape - Overview

- Initialize the shape with ellipsoid.
- Render '*pseudo samples*' with different viewpoints and lighting conditions.
- GAN inversion is applied to these samples to obtain the '*projected samples*'.
- *'Projected samples'* are used as the ground truth of the rendering process to optimize the 3D shape.
- Iterative training to progressively refine the shape.



3D Generation Results

Without any 2D keypoint or 3D annotations

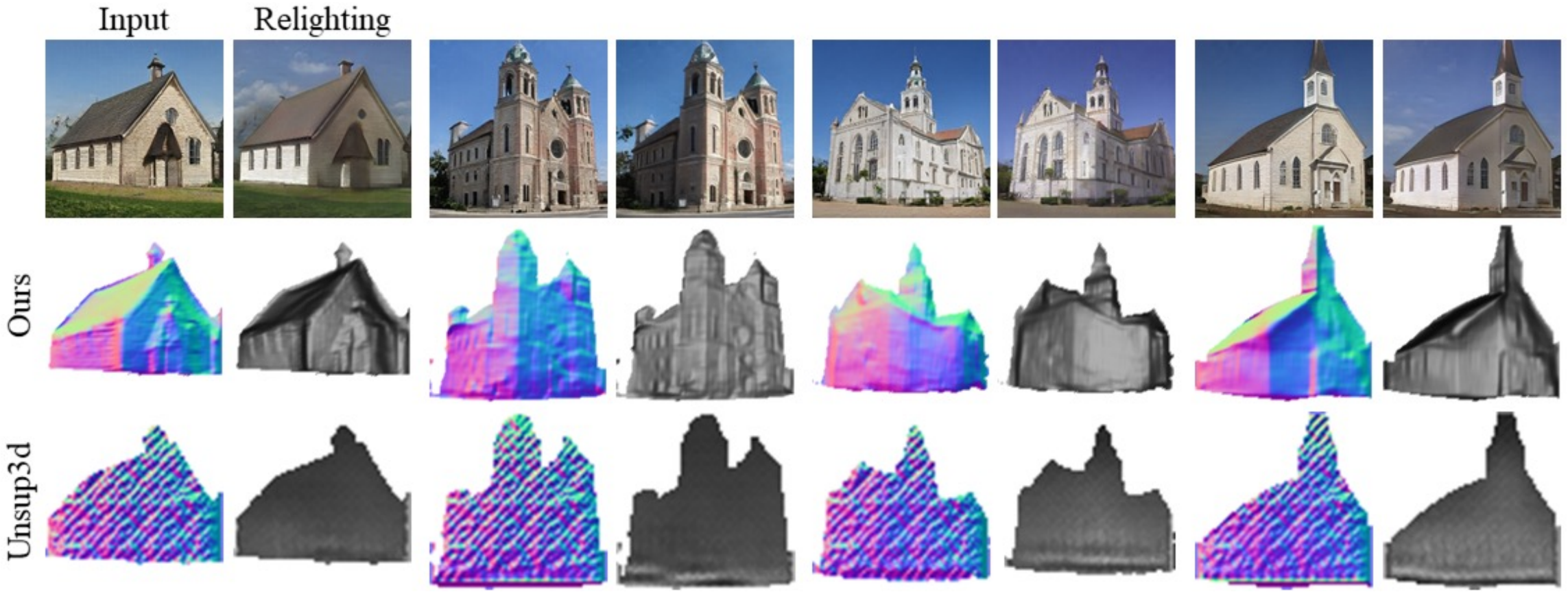
Unsupervised 3D shape reconstruction from unconstrained 2D images

Without symmetry assumption

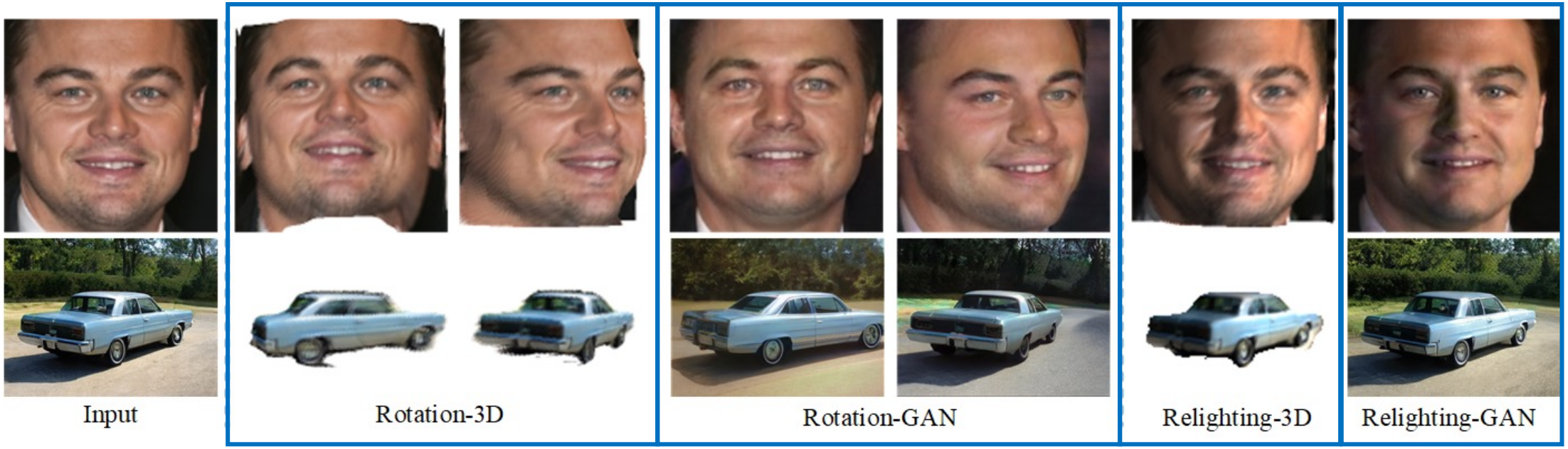
Work on many object categories such as huma faces, cars, buildings, etc.

Shangzhe Wu, et al., Unsupervised learning of probably symmetric deformable 3D objects from images in the wild. In *CVPR*, 2020

3D Generation Results



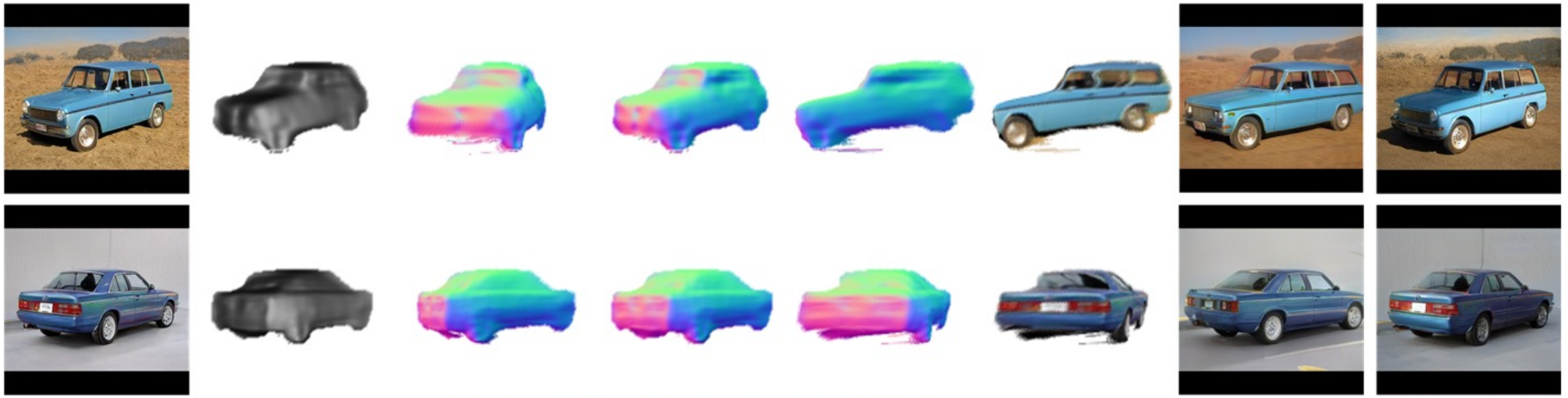
3D-aware Image Manipulation



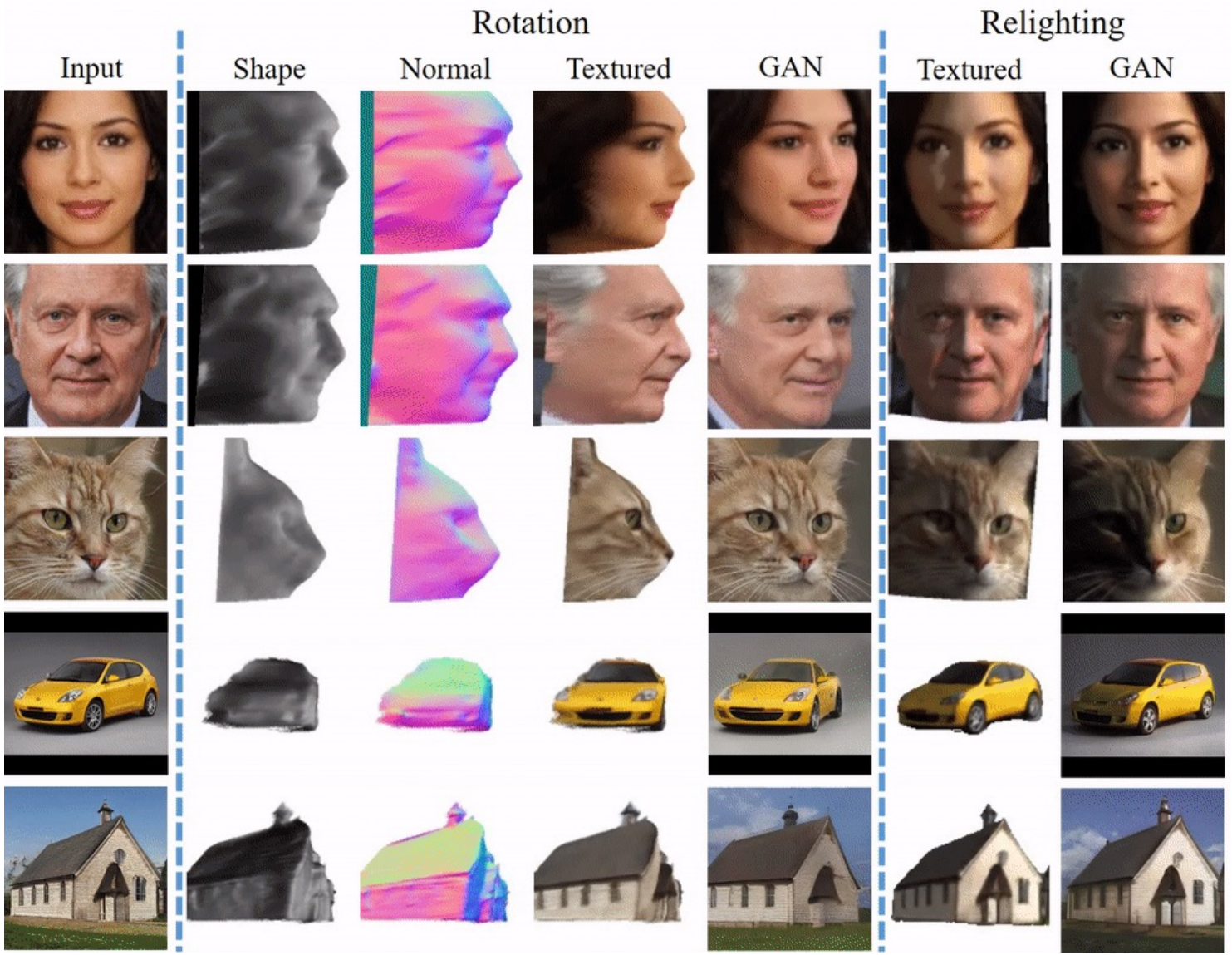
- Effect-3D: Rendered using the reconstructed 3D shape and albedo.
- Effect-GAN: project Effect-3D on the GAN image manifold using the trained encoder *E*.

More Results

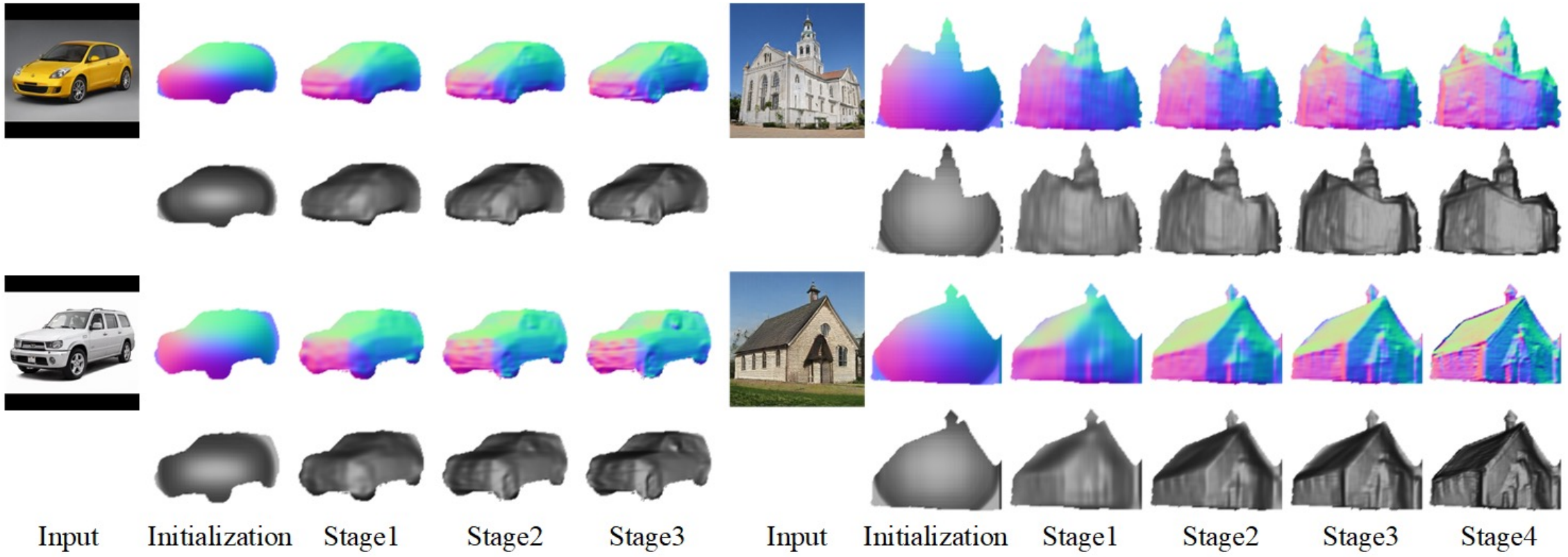
More Results



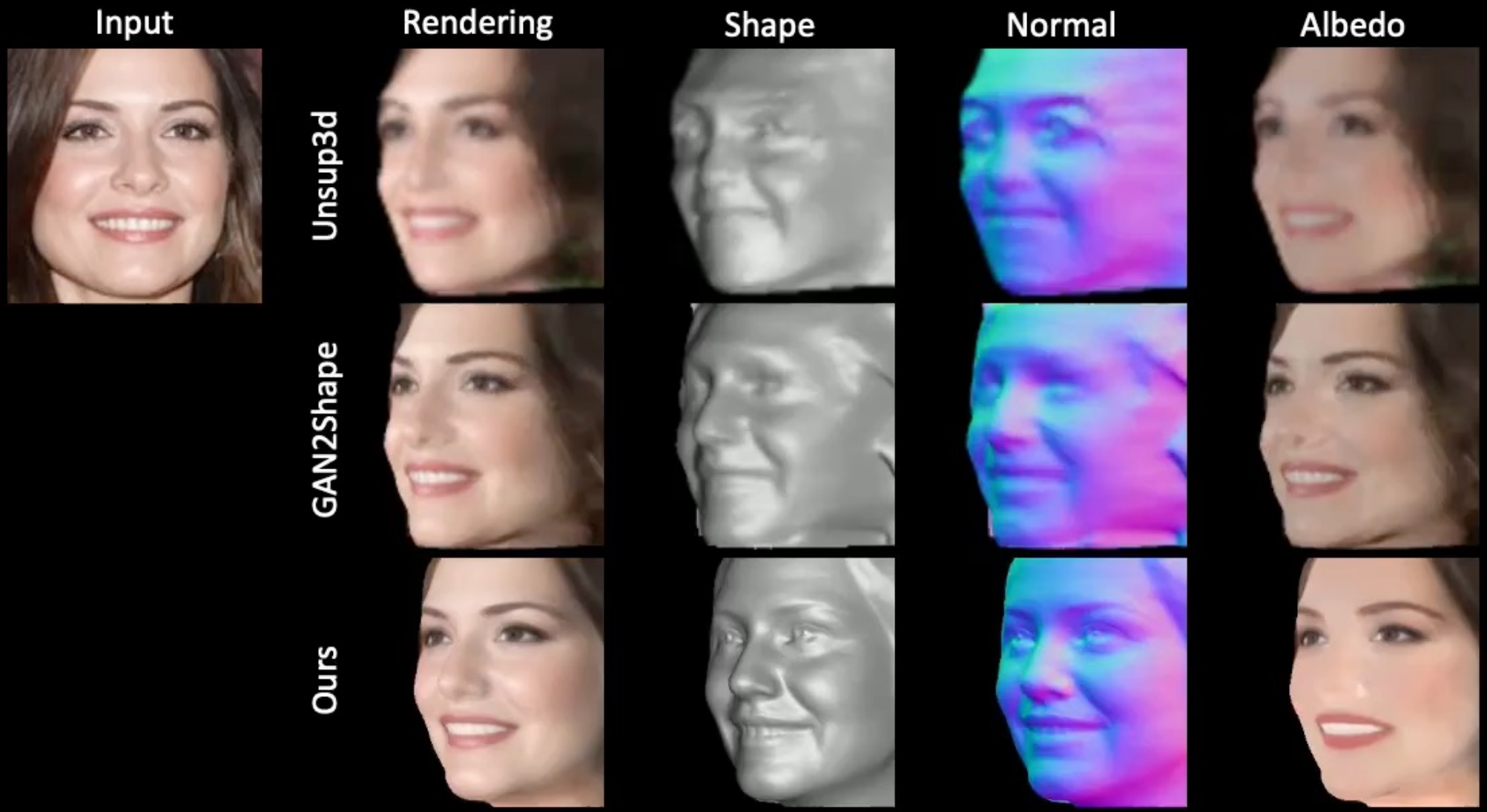
Demo


Rotation
Relighting
Input
Shape
Normal
Textured
GAN
Textured
GAN

Effects of Iterative Training

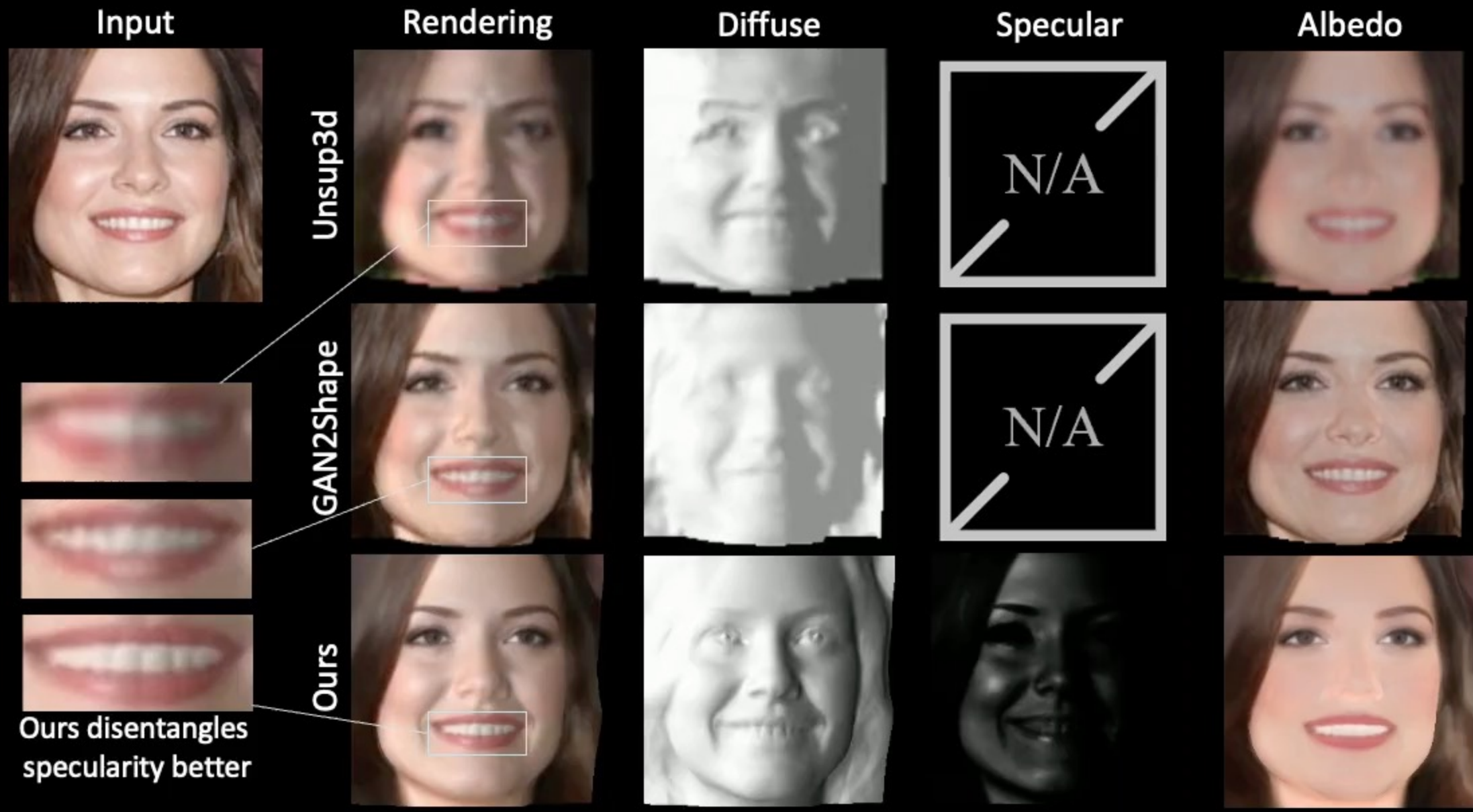


Qualitative Comparison on CelebA: Rotation



Pan et al, GAN2X: Non-Lambertian Inverse Rendering of Image GANs, 3DV

Qualitative Comparison on CelebA: Relighting

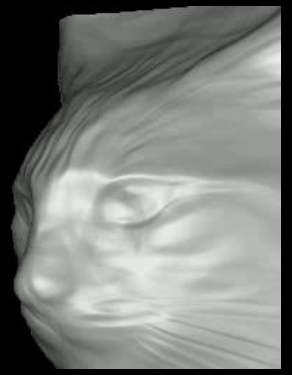
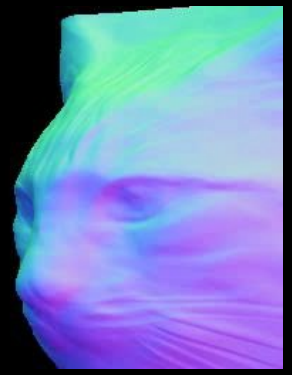


Input Rendering Shape Normal Albedo

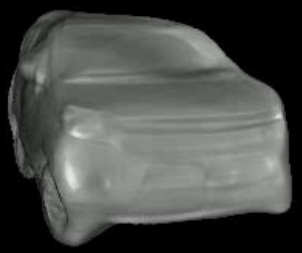
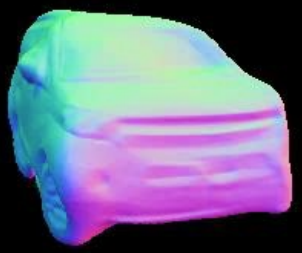








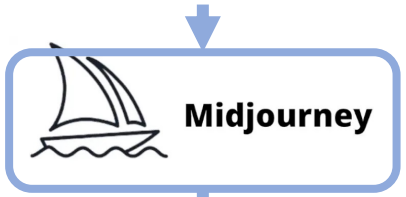

Input Rendering Diffuse Specular Albedo

Drag Your GAN

Xingang Pan, et al., Drag Your GAN: Interactive Point-based Manipulation on the Generative Image Manifold. SIGGRAPH 2023

What's Missing in Image Synthesis

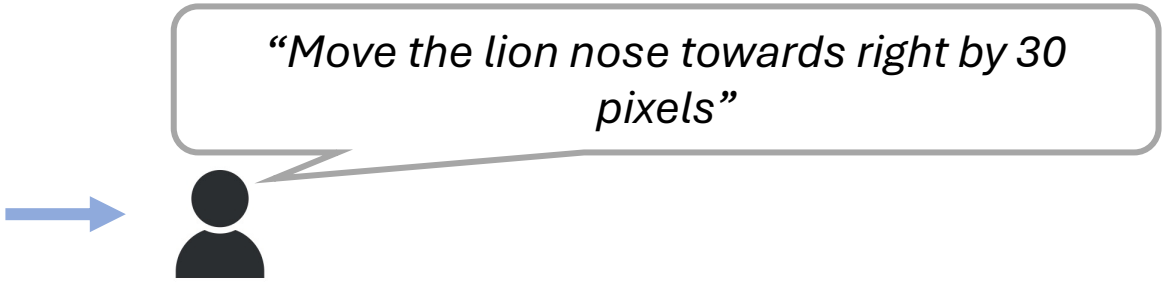
"A stunning portrait of a male lion"



What if the user wants to …

- **Change the pose (e.g., rotate head)?**
- **Increase/reduce the animal size?**
- **Move the animal?**
- **Change the expression?**
- **……**

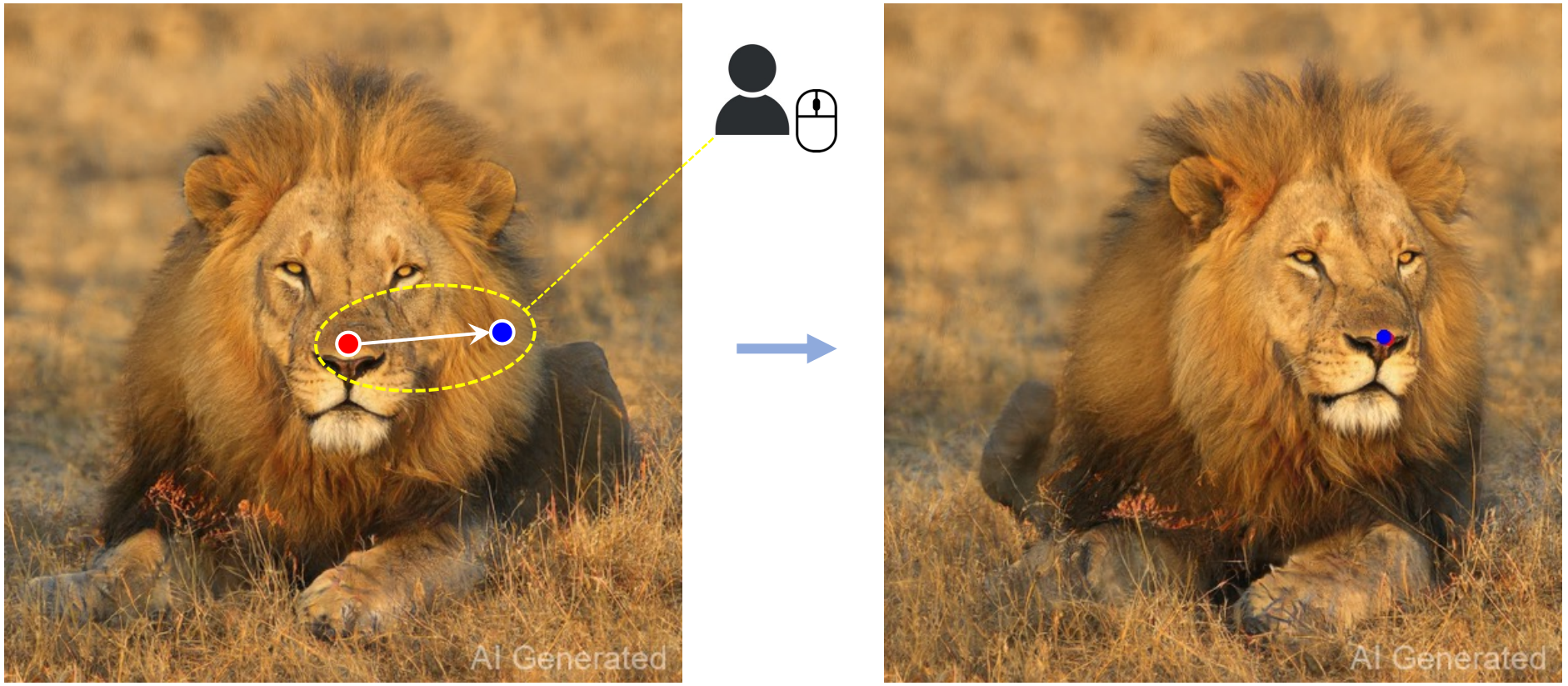
How Should We Control Spatial Attributes?



Limitations

- The language model need to understand all possible spatial editing commands
- It’s hard for language model to understand the precise distance (30 pixels) in the image

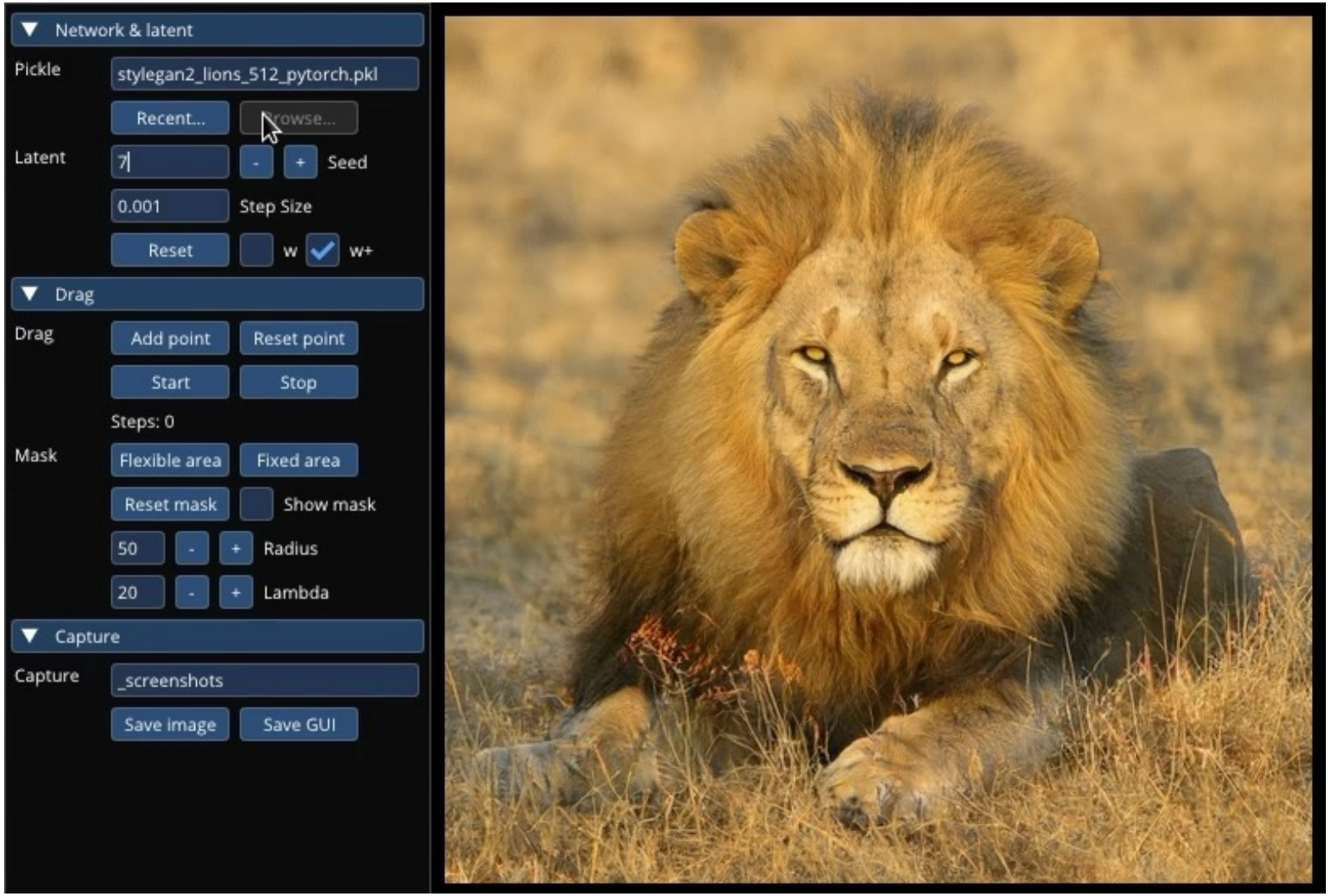
Interactive Point Dragging



Advantages

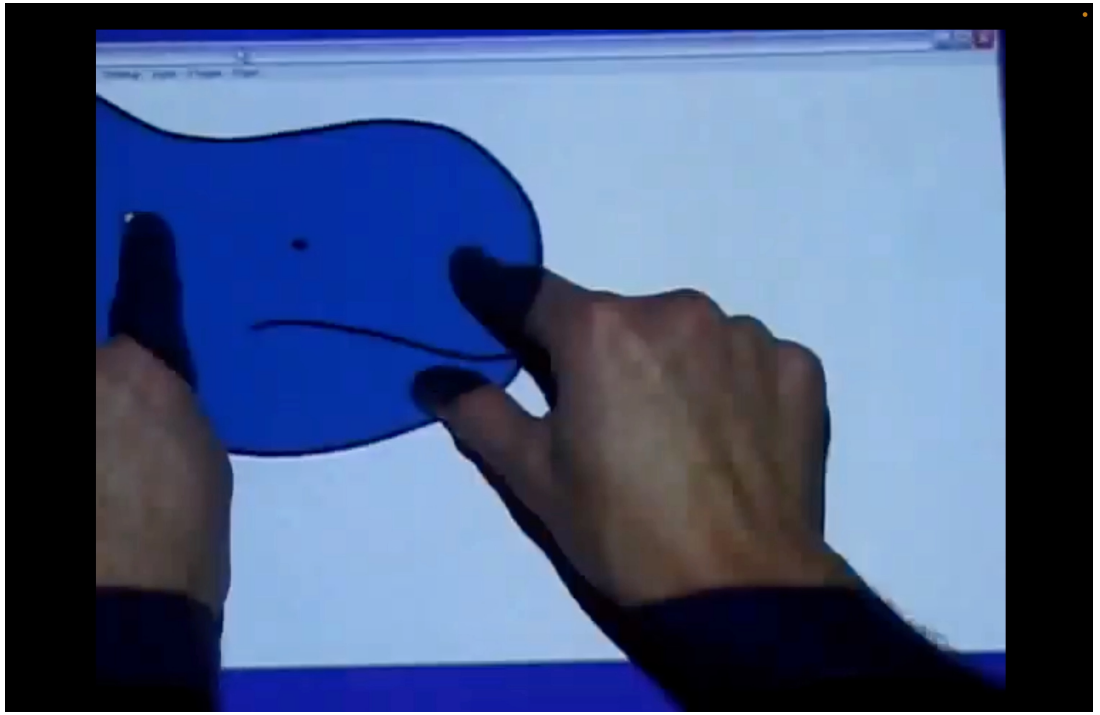
- Simple
- Precise
- Flexible
- Generic

Interactive Point Dragging



Conventional Shape Deformation

As-Rigit-As-Possible Shape Manipulation

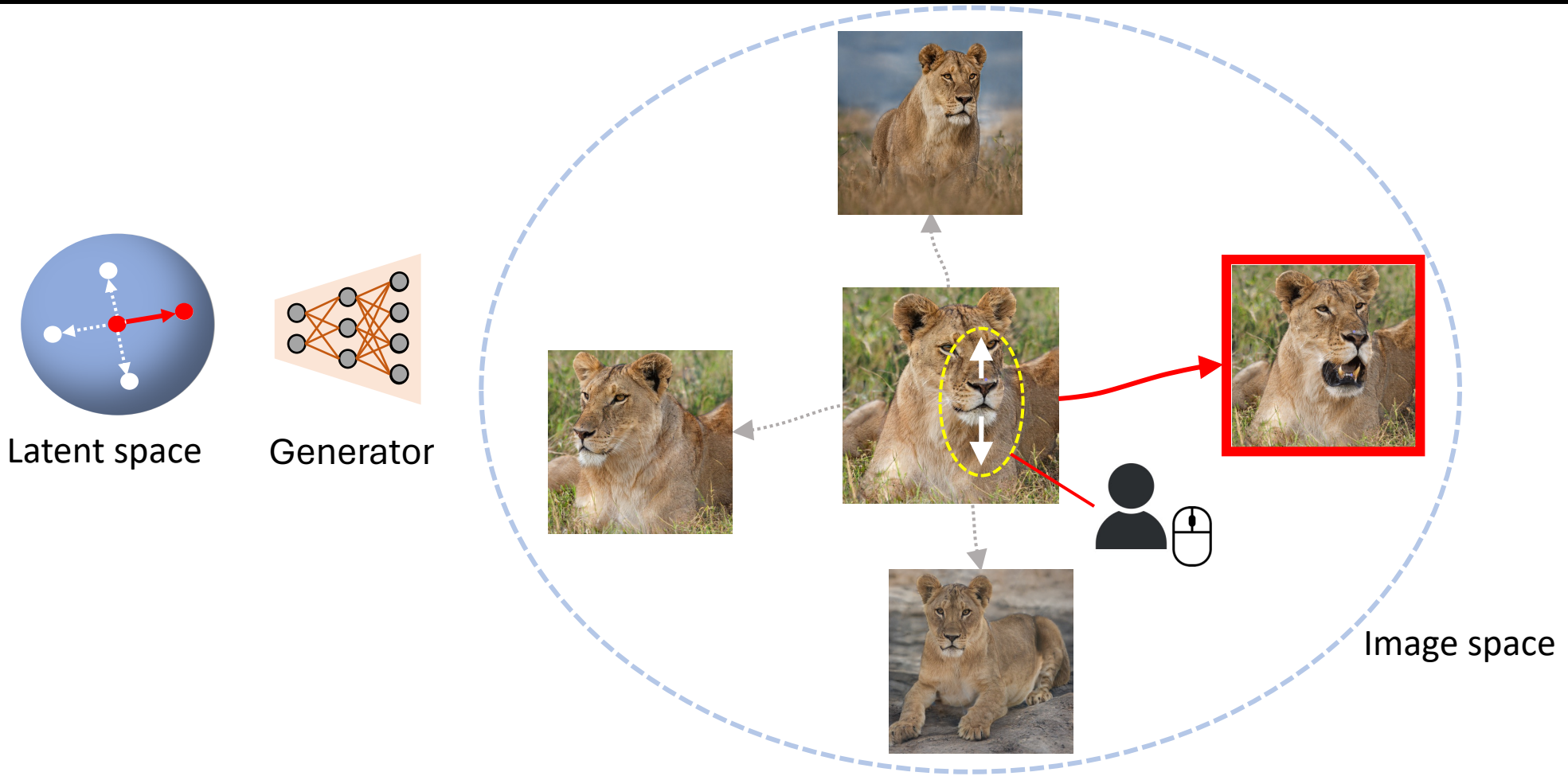


Limitations:

- Assumes uniform rigidity
- Cannot hallucinate new content

Igarashi, et al, As-Rigid-As-Possible Shape Manipulation, TOG 2005

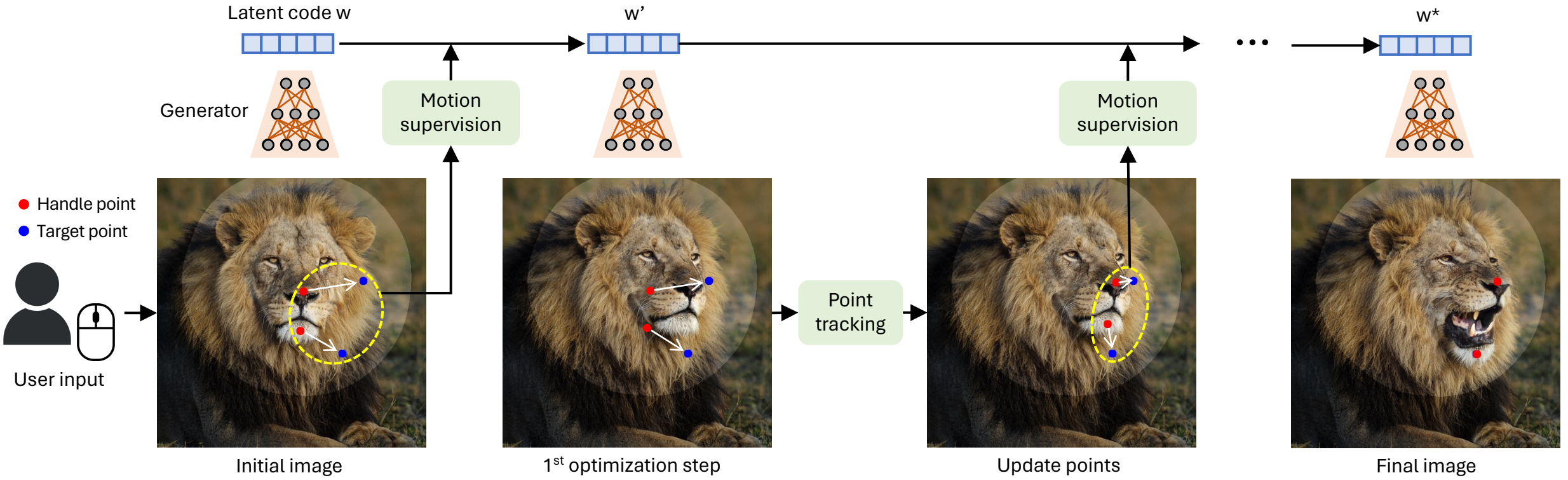
Motivation



GAN captures different variations of an object (pose, size, expression, etc)

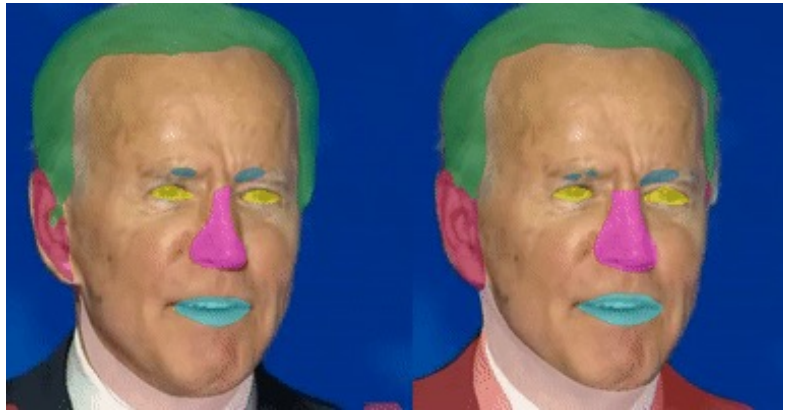
We aims to traverse along the path that follows the dragging operation

Method Overview



Discriminative GAN Features

Few shot segmentation



RepurposeGAN [Rewatbowornwong et al, CVPR2021]

Few shot segmentation

DatasetGAN [Zhang et al, CVPR2021]

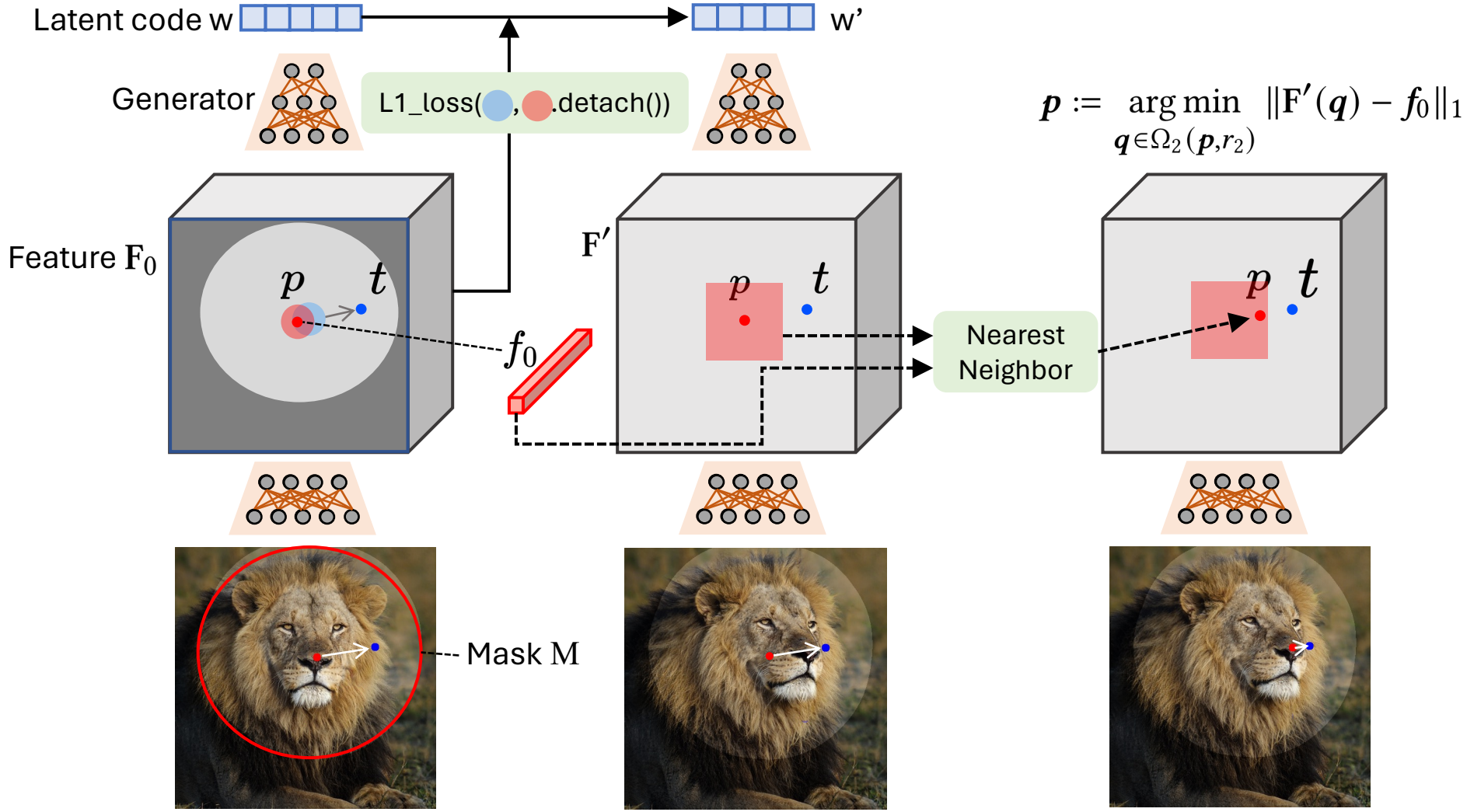
Unsupervised segmentation

Labels4Free [Abdal et al, ICCV2021]

Dense Correspondence

Dual Deformation Field [Lan et al, arXiv, 2022]

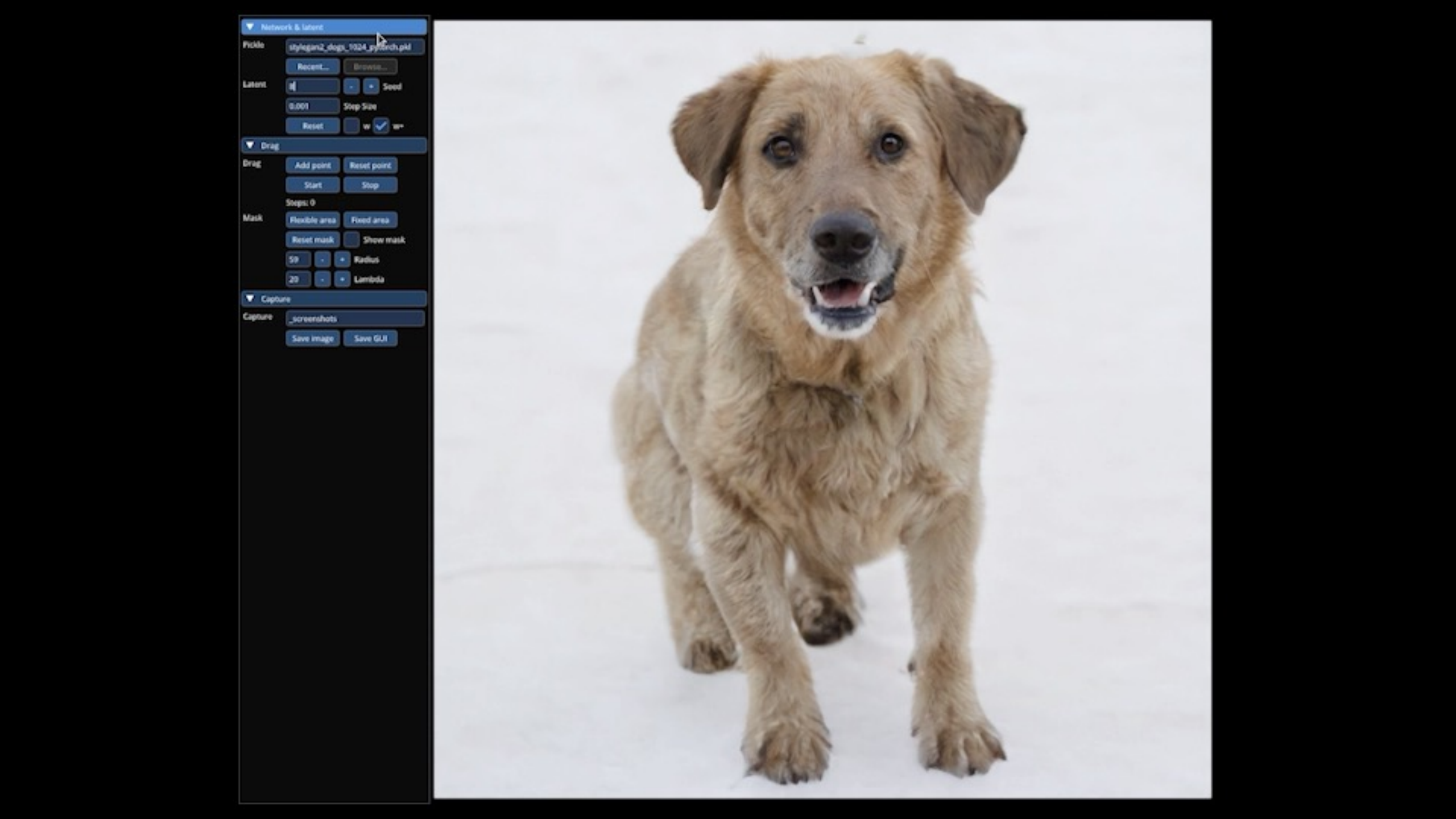
Motion Sup. and Point Tracking in GAN Features



$$\boldsymbol{p} := \arg\min_{\boldsymbol{q}\in\Omega_2(\boldsymbol{p},r_2)} \|\mathbf{F}'(\boldsymbol{q}) - \boldsymbol{f}_0\|_1$$

$$\mathcal{L} = \sum_{\boldsymbol{q}\in\Omega_1(\boldsymbol{p},r_1)} \|\mathbf{F}(\boldsymbol{q}) - \mathbf{F}(\boldsymbol{q}+\boldsymbol{d})\|_1 + \lambda\|(\mathbf{F}-\mathbf{F}_0)\cdot(1-\mathbf{M})\|_1$$

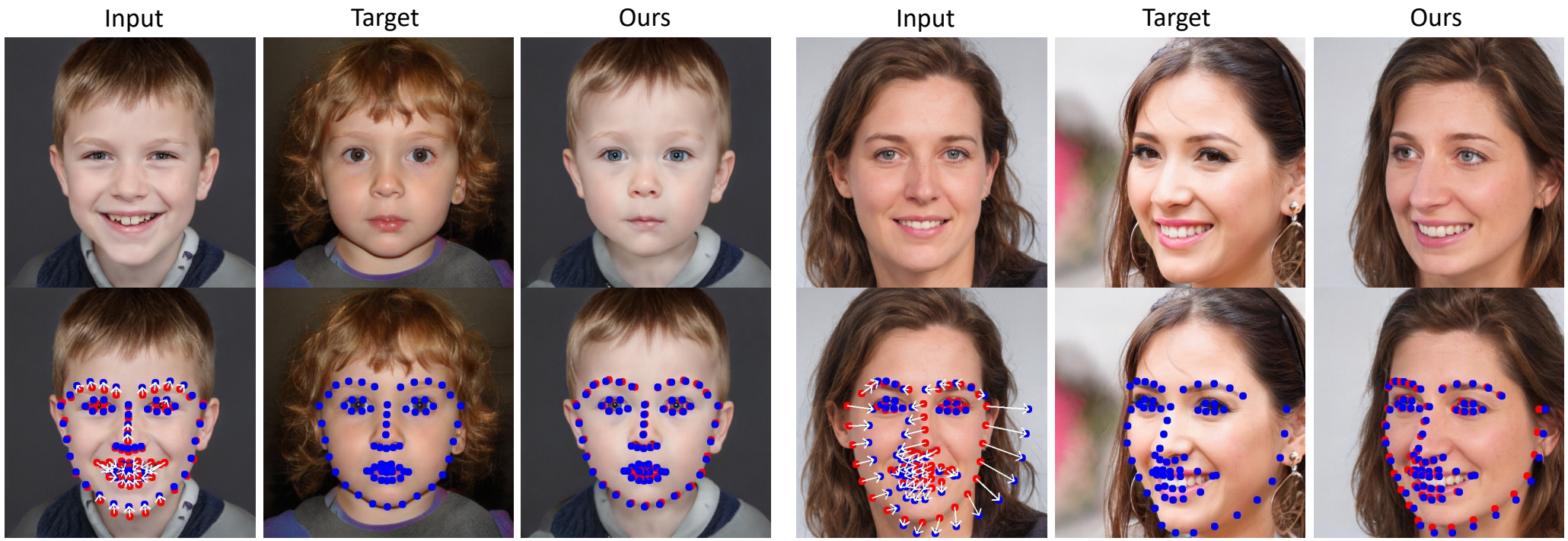
$$\boldsymbol{d} = \frac{\boldsymbol{t}-\boldsymbol{p}}{\|\boldsymbol{t}-\boldsymbol{p}\|_2}$$


Network & latent
Pickle
stylegan2_dogs_1024_pytorch.pkl
Recent...
Browse...
Latent
0
Seed
0.001
Step Size
Reset
w
w+
Drag
Drag
Add point
Reset point
Start
Stop
Steps: 0
Mask
Flexible area
Fixed area
Reset mask
Show mask
59
Radius
20
Lambda
Capture
Capture
_screenshots
Save image
Save GUI

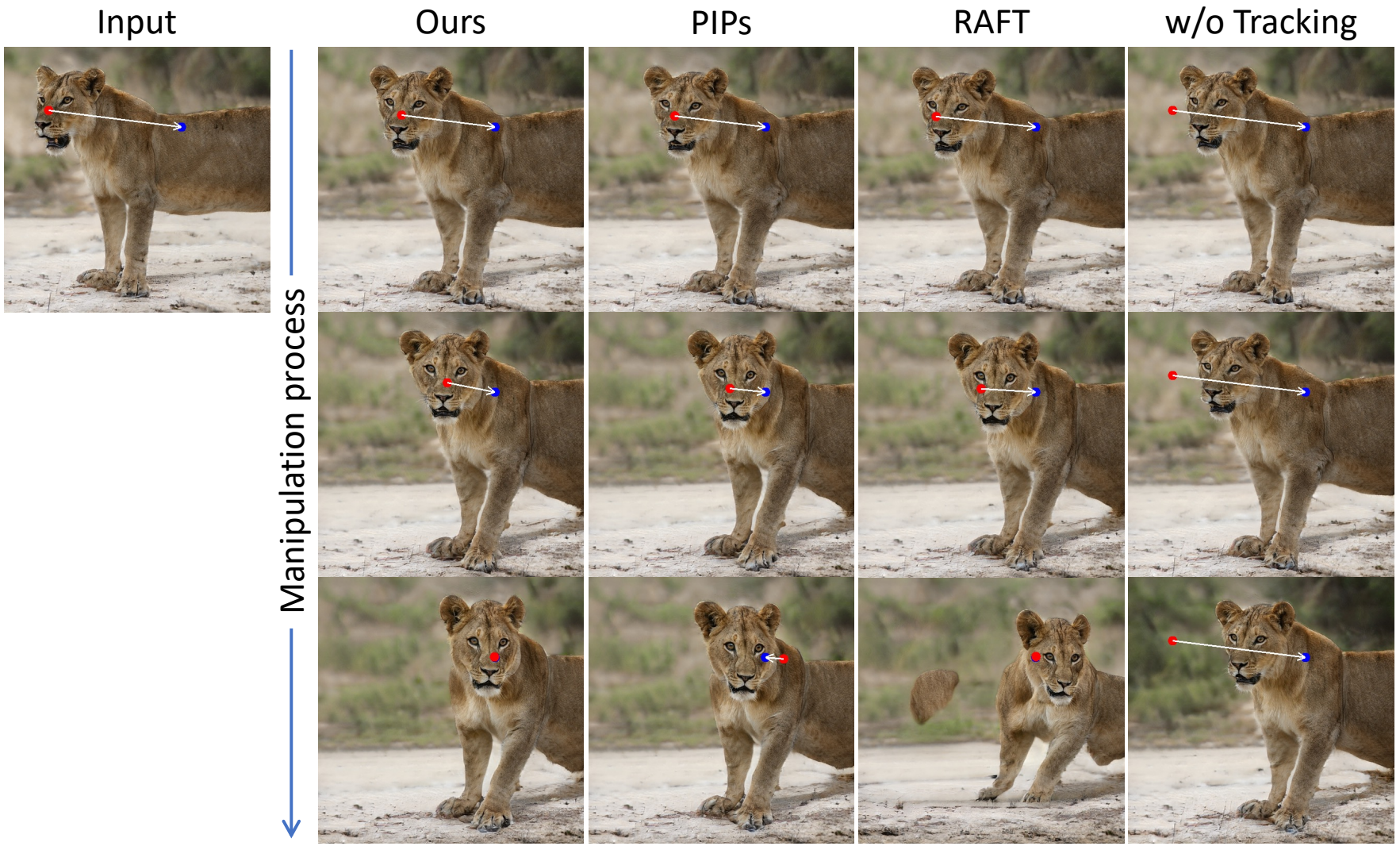
Qualitative Comparison

Yuki Endo, User-Controllable Latent Transformer for StyleGAN Image Layout Editing, Pacific Graphics 2022

Face Landmark Manipulation

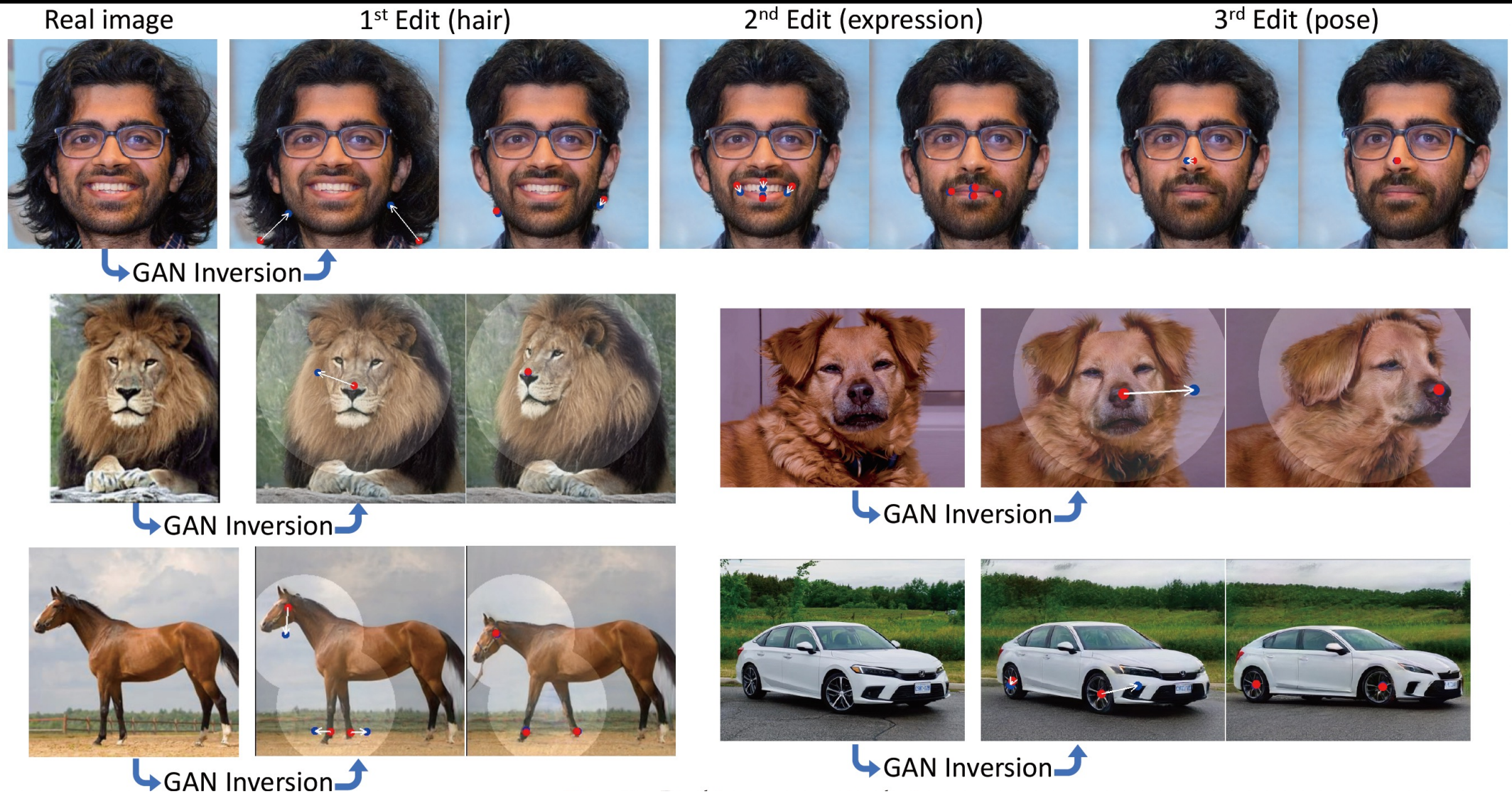


Point Tracking



Harley et al, Particle Video Revisited: Tracking Through Occlusions Using Point Trajectories, ECCV2022
Zachary et al, RAFT: Recurrent All-Pairs Field Transforms for Optical Flow, ECCV2020

Real Image Manipulation

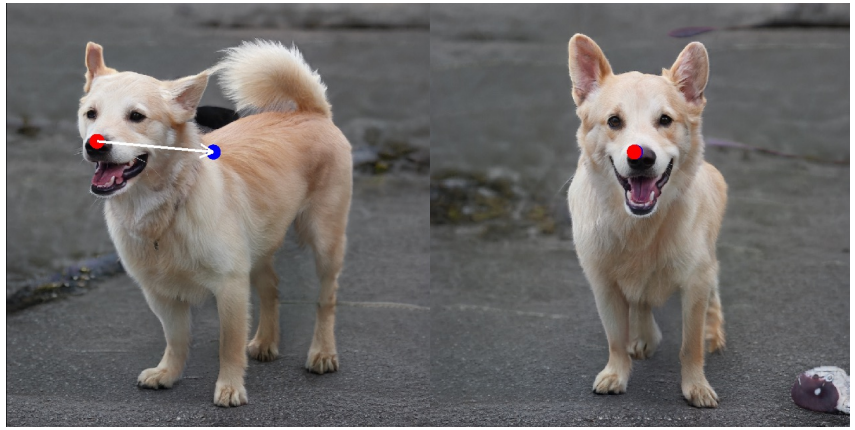


Pivotal Tuning for Latent-based editing of Real Images, Roich et al, ACM TOG 2022

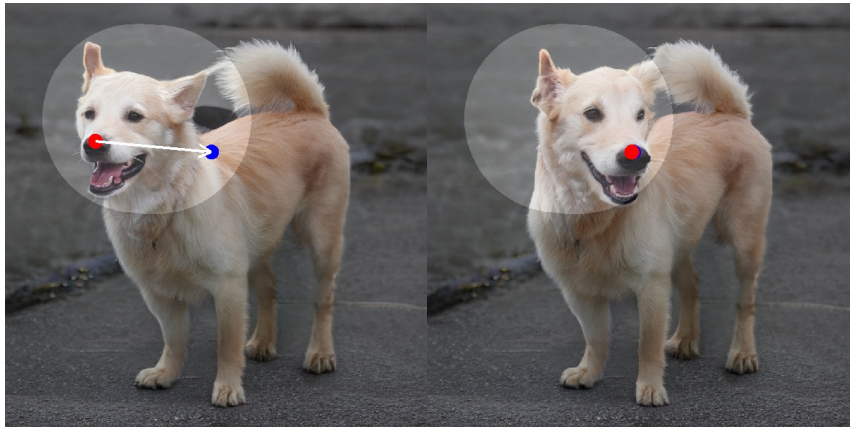
Discussions

Effects of Mask

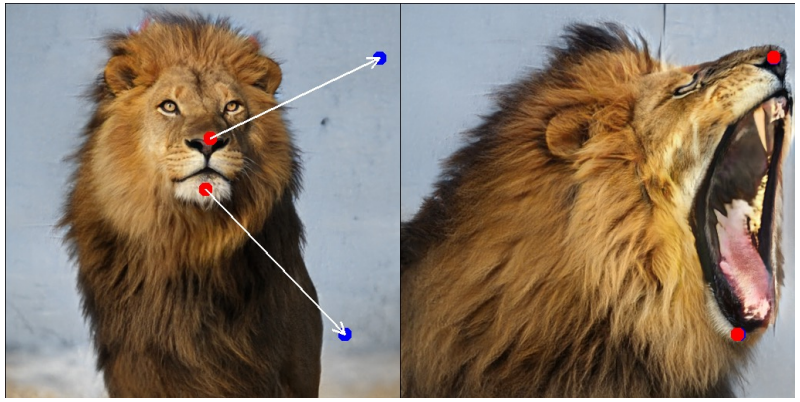
w/o mask



w/ mask



Out-of-distribution Manipulations



Thank you!