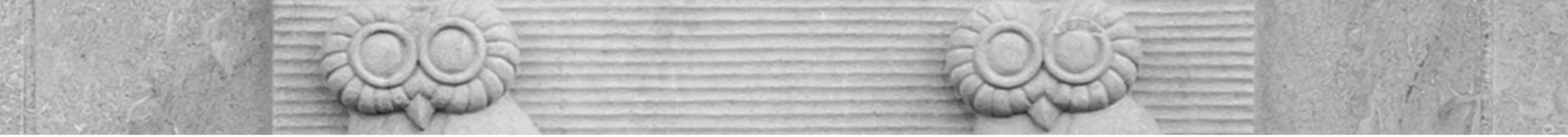
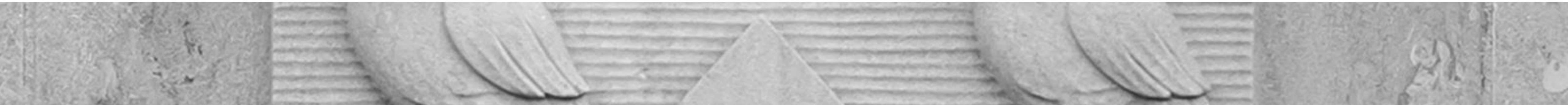


# Deep Learning for Vision & Language

Segmentation, AutoEncoders, Variational AutoEncoders, Introduction to Diffusion Models

RICE UNIVERSITY



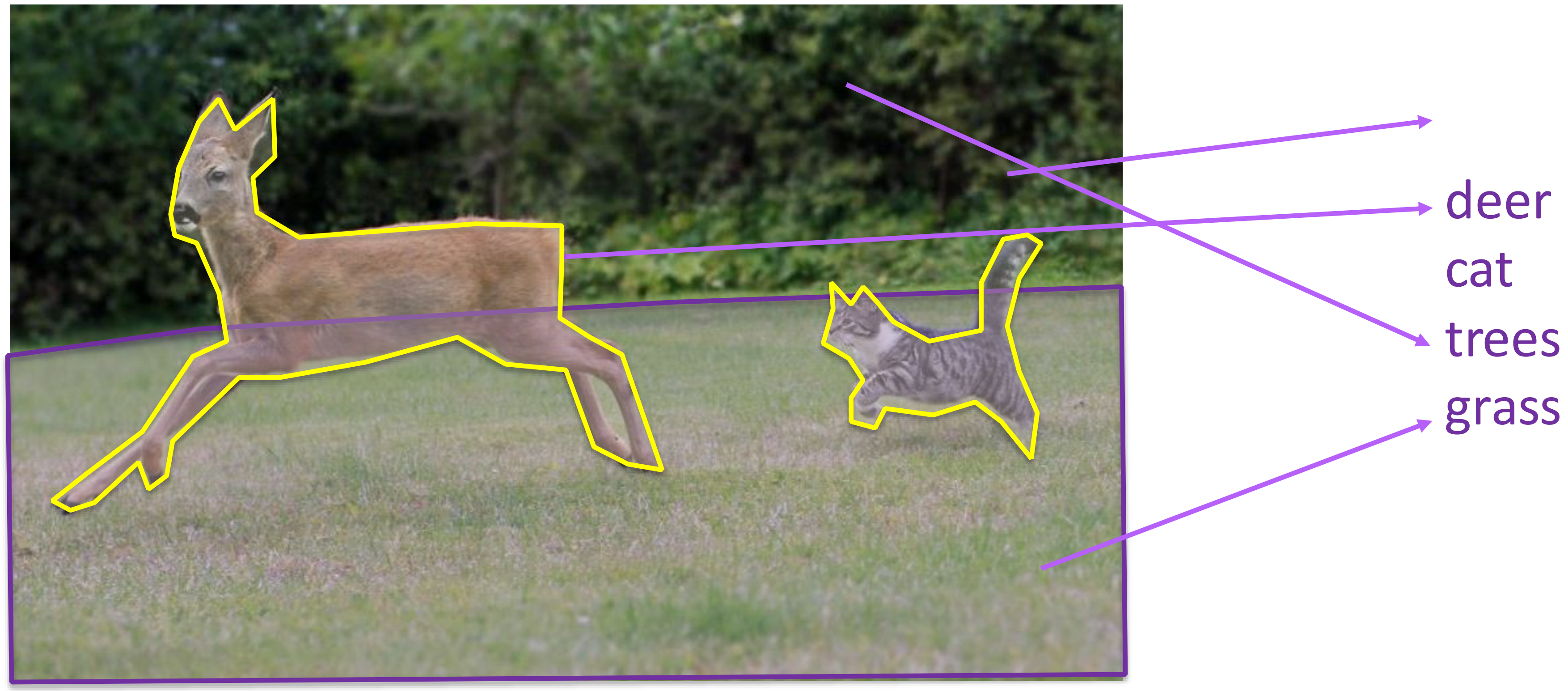
## Final Project

- I am still giving feedback on the project proposal. There’s no grade for the project proposal. There’s also no grade for class attendance. See course syllabus. I won’t change course syllabus in the middle of the course. I aim to send feedback to everyone before Spring Break.
- I will provide you an opportunity to provide a Project Progress Report so you have an idea of my expectations and grade and avoid “surprises”.
- Final project deliverables is worth 60% of the class divided as follows:
  - Originality: 10pts
  - Technical accuracy: 10pts
  - Presentation: 10pts
  - Results: 10pts

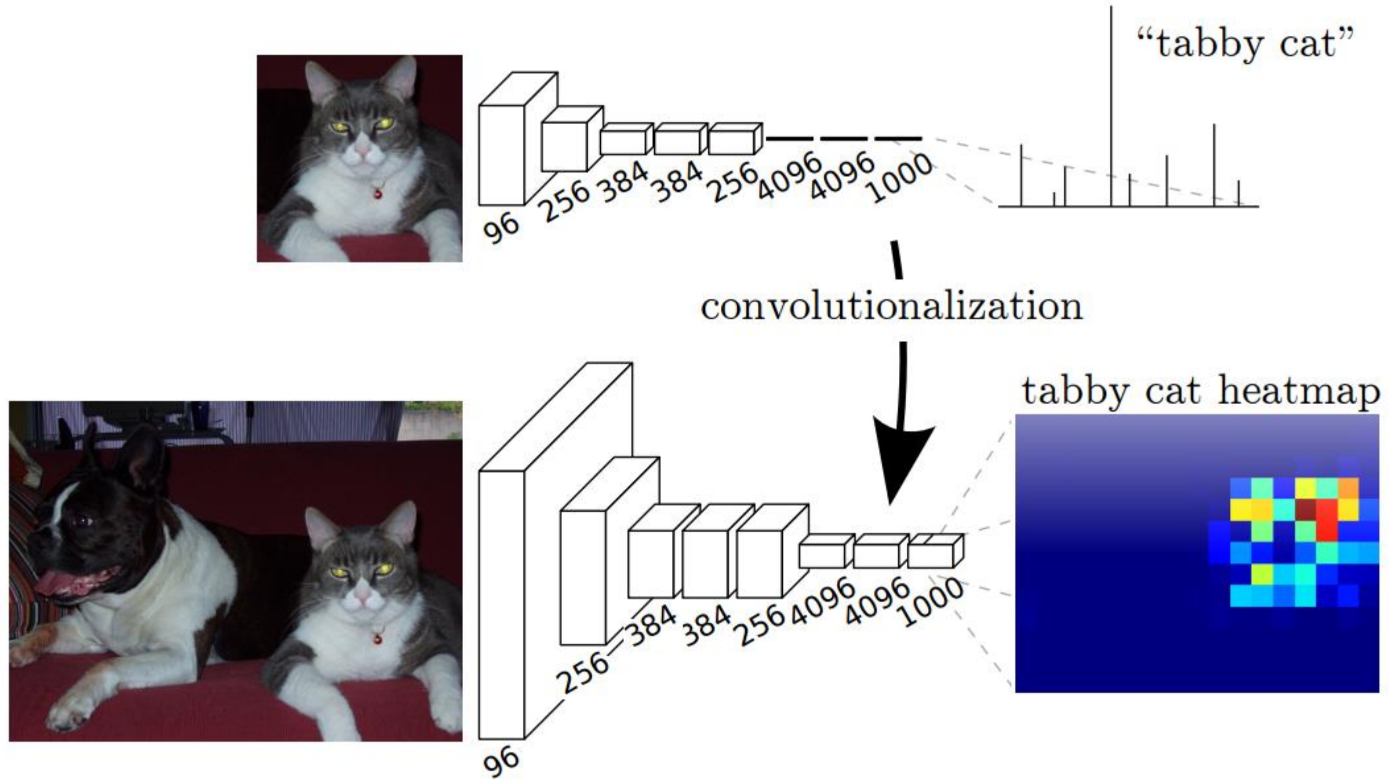
## Final Project

- Originality: Is the idea of the project original itself? Is the original idea of the project standard but is any of the experiments original? [10pts]
- Technical accuracy: Is your final project report accurate? Are all the terms used appropriately and are the mathematical expressions used to explain your model/loss/optimization accurate?
- Presentation: Is your final report publication-ready? Could your final project report be published as a technical report? I don't want to see unprofessional practices that you would not see on a technical paper. Common mistakes: Pixelated images, axis on output figures that don't warrant axes, etc.
- Results: Are your results good? I can judge this by using metrics such as accuracy, BLEU, retrieval@K, human surveys, and I can also judge by looking at any input-output results that you include. I want to SEE you did well.

# Semantic Segmentation / Image Parsing



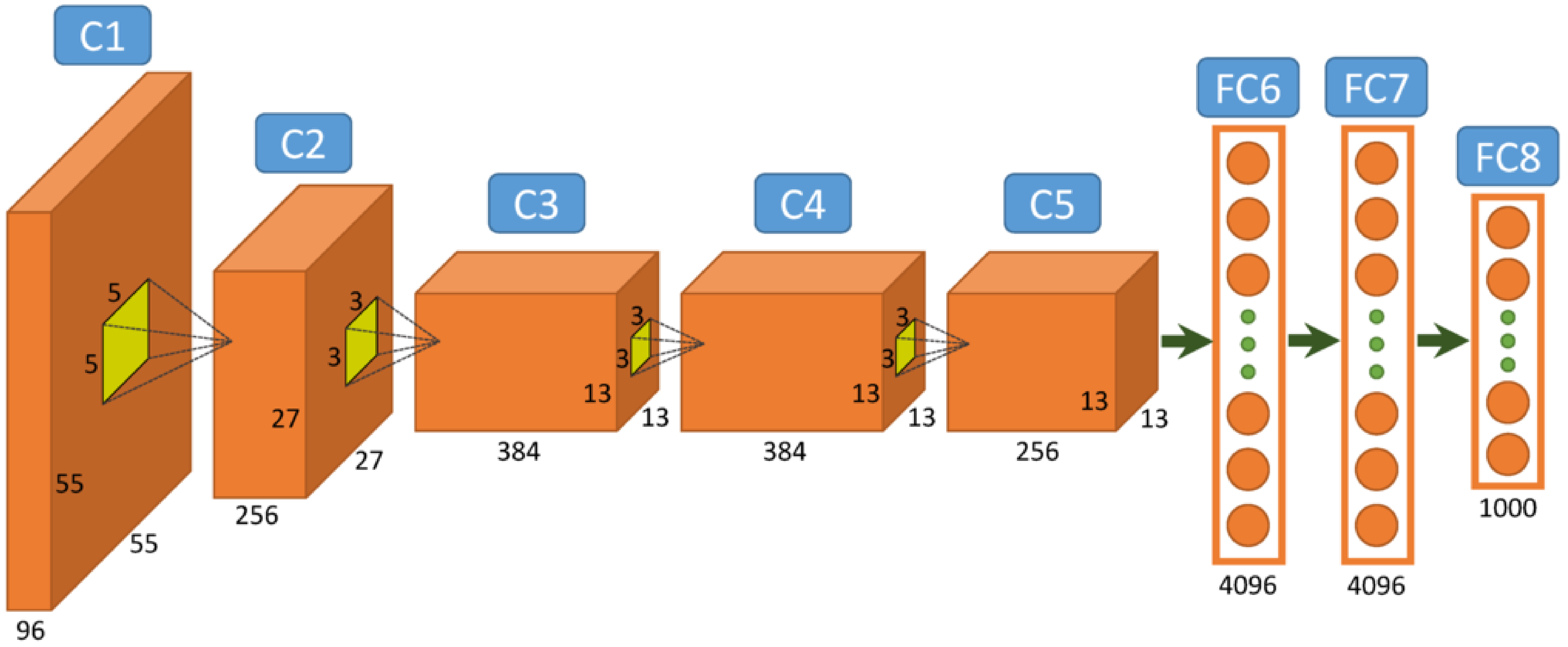
# Idea 1: Convolutionalization



However resolution of the segmentation map is low.

https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf

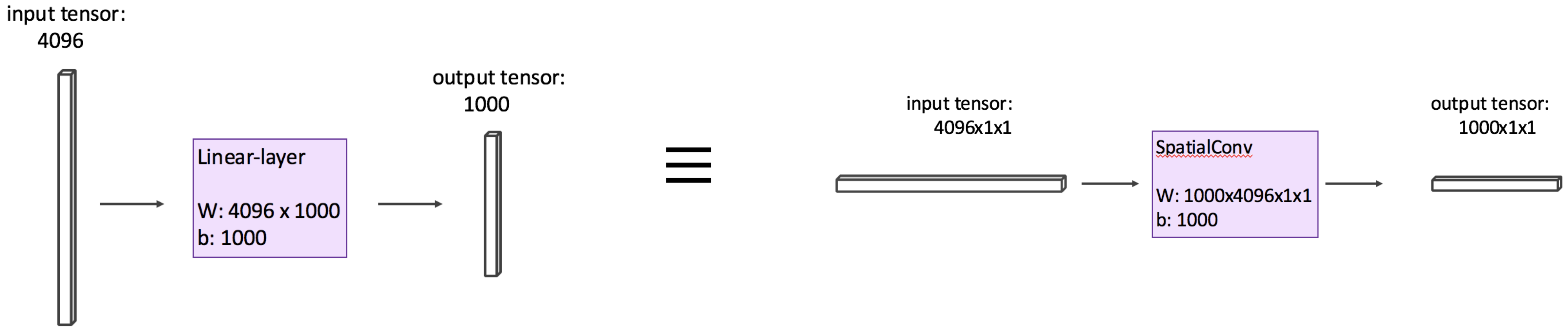
## Alexnet



https://www.saagie.com/fr/blog/object-detection-part1

## Idea 1: Convolutionalization

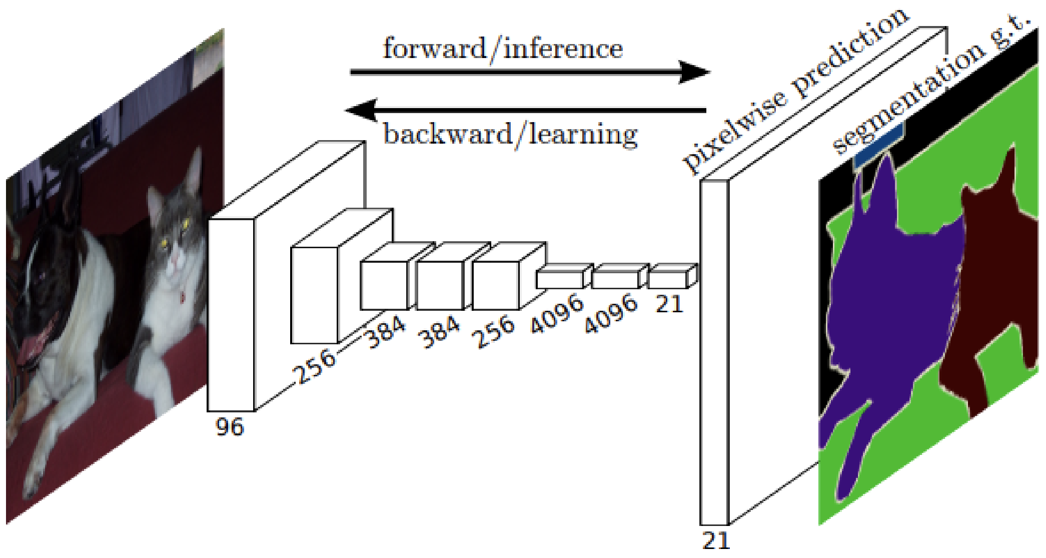
nn.Linear(4096, 1000) == nn.Conv2D(4096, 1000, kernel_size = 1, stride = 1)



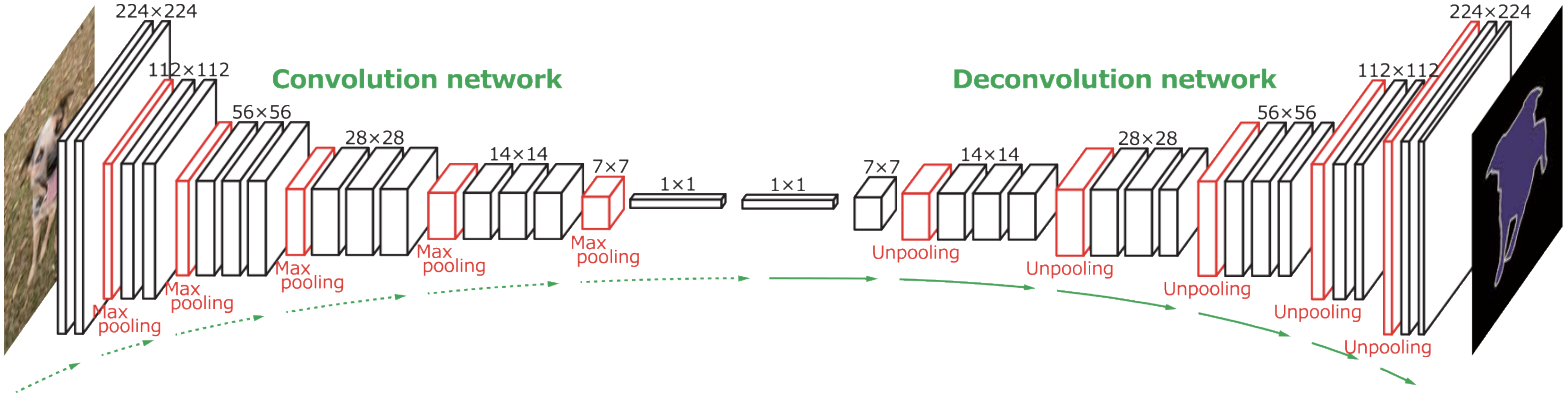
# Fully Convolutional Networks (CVPR 2015)

## Fully Convolutional Networks for Semantic Segmentation

Jonathan Long* Evan Shelhamer* Trevor Darrell
UC Berkeley
{jonlong,shelhamer,trevor}@cs.berkeley.edu



# Idea 2: Up-sampling Convolutions or ”Deconvolutions” or Transposed Convolutions

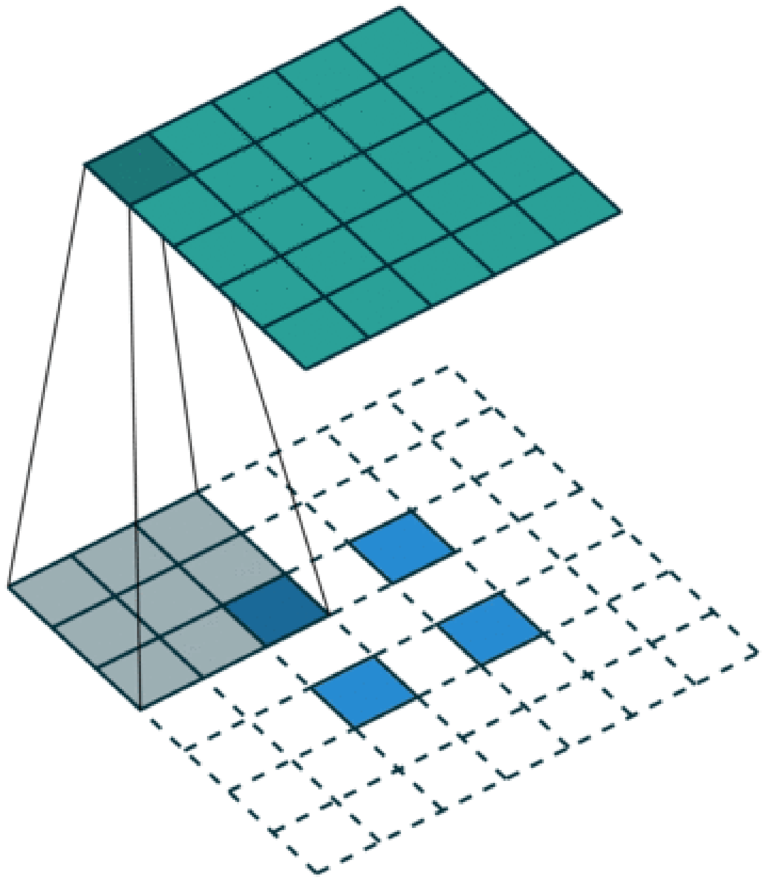


**Learning Deconvolution Network for Semantic Segmentation**

Hyeonwoo Noh Seunghoon Hong Bohyung Han
Department of Computer Science and Engineering, POSTECH, Korea
{hyeonwoonoh_,maga33,bhhan}@postech.ac.kr

http://cvlab.postech.ac.kr/research/deconvnet/

## Idea 2: Up-sampling Convolutions or ”Deconvolutions” or Transposed Convolutions



https://github.com/vdumoulin/conv_arithmetic

## Idea 2: Up-sampling Convolutions or ”Deconvolutions”

Deconvolutional Layers

Upconvolutional Layers

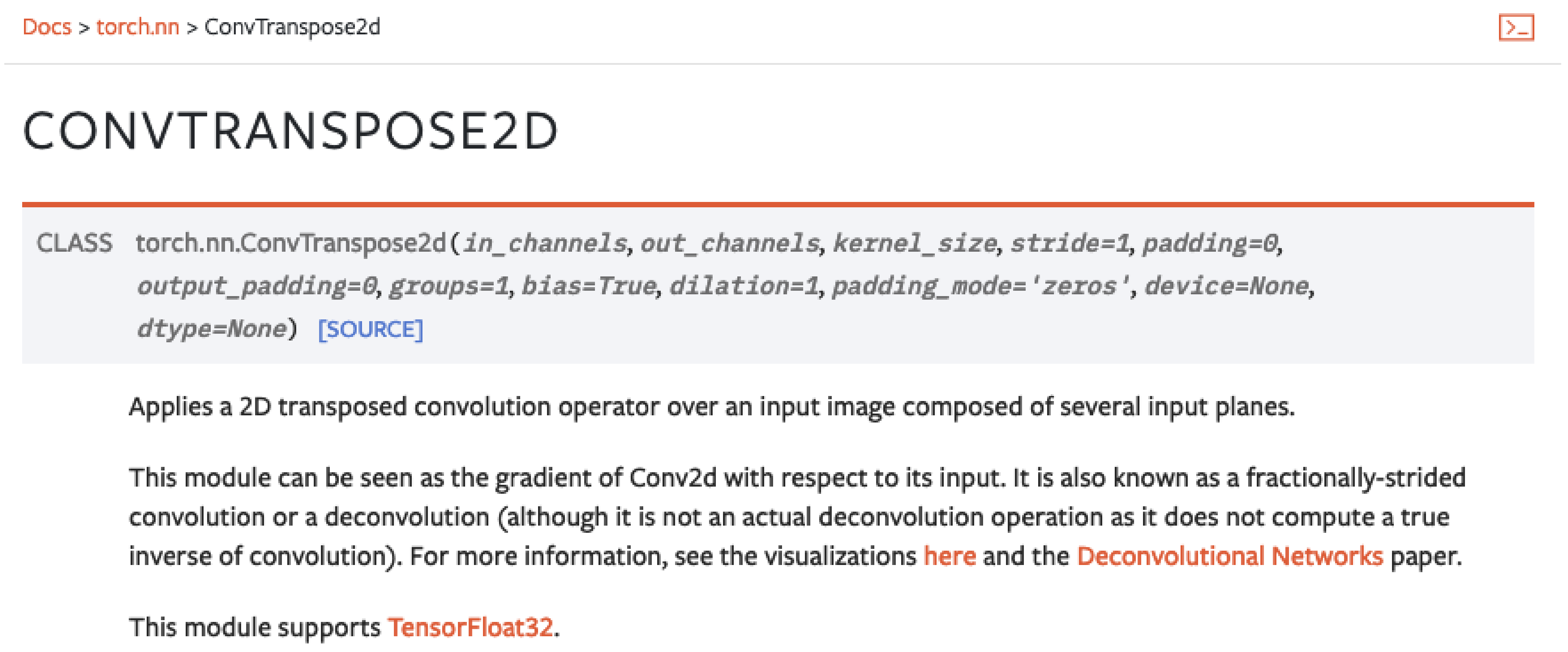
Backwards Strided
Convolutional Layers

Fractionally Strided
Convolutional Layers

Transposed
Convolutional Layers

Spatial Full
Convolutional Layers

# Pytorch



Docs > torch.nn > ConvTranspose2d

## CONVTRANSPOSE2D

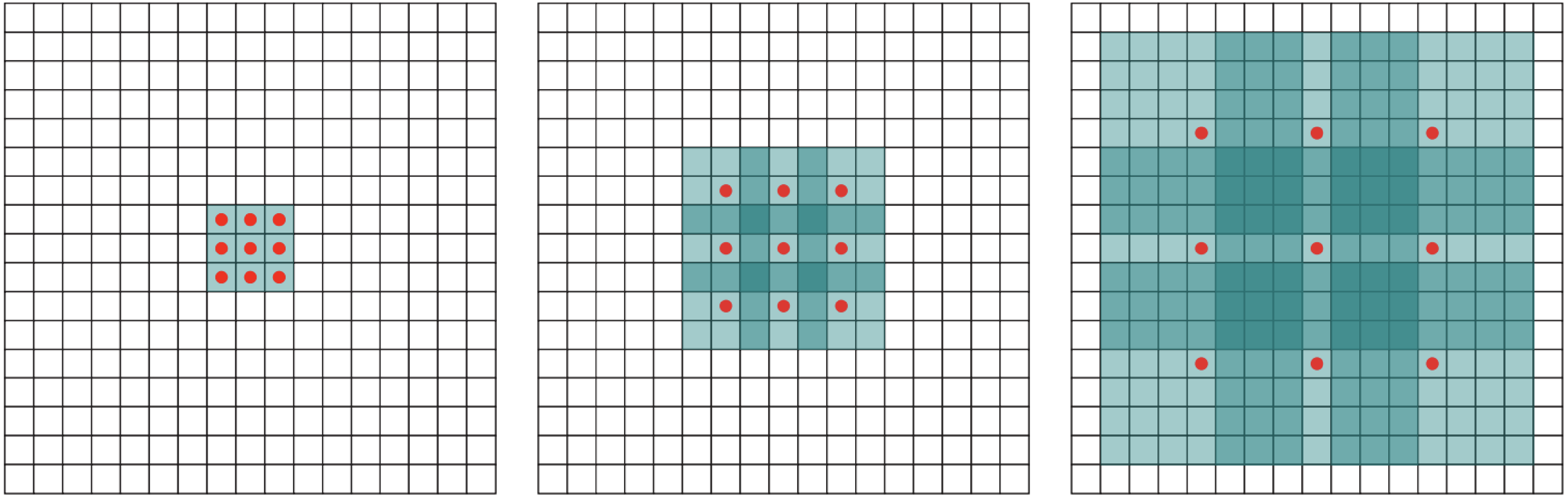
CLASS torch.nn.ConvTranspose2d(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *bias=True*, *dilation=1*, *padding_mode='zeros'*, *device=None*, *dtype=None*) [SOURCE]

Applies a 2D transposed convolution operator over an input image composed of several input planes.

This module can be seen as the gradient of Conv2d with respect to its input. It is also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation as it does not compute a true inverse of convolution). For more information, see the visualizations here and the Deconvolutional Networks paper.

This module supports TensorFloat32.

## Idea 3: Dilated Convolutions



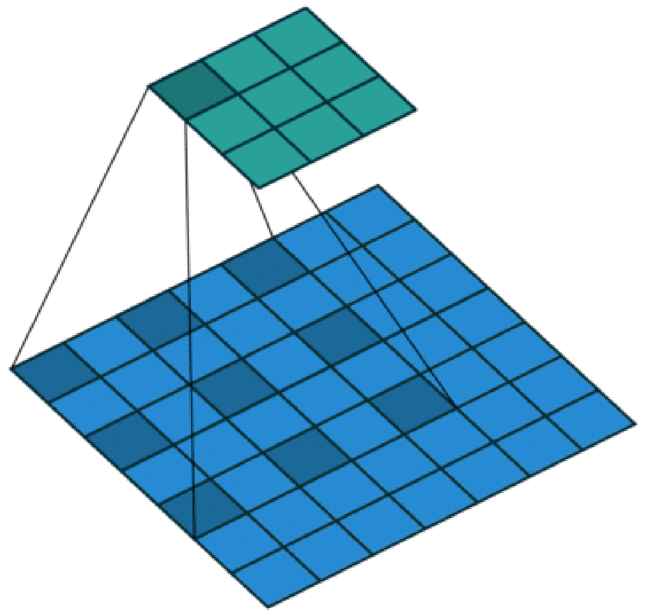
MULTI-SCALE CONTEXT AGGREGATION BY DILATED CONVOLUTIONS

**Fisher Yu**
Princeton University

**Vladlen Koltun**
Intel Labs

ICLR 2016

## Idea 3: Dilated Convolutions



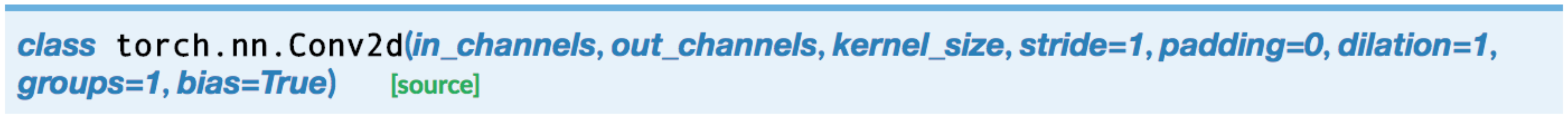
MULTI-SCALE CONTEXT AGGREGATION BY DILATED CONVOLUTIONS

**Fisher Yu**
Princeton University

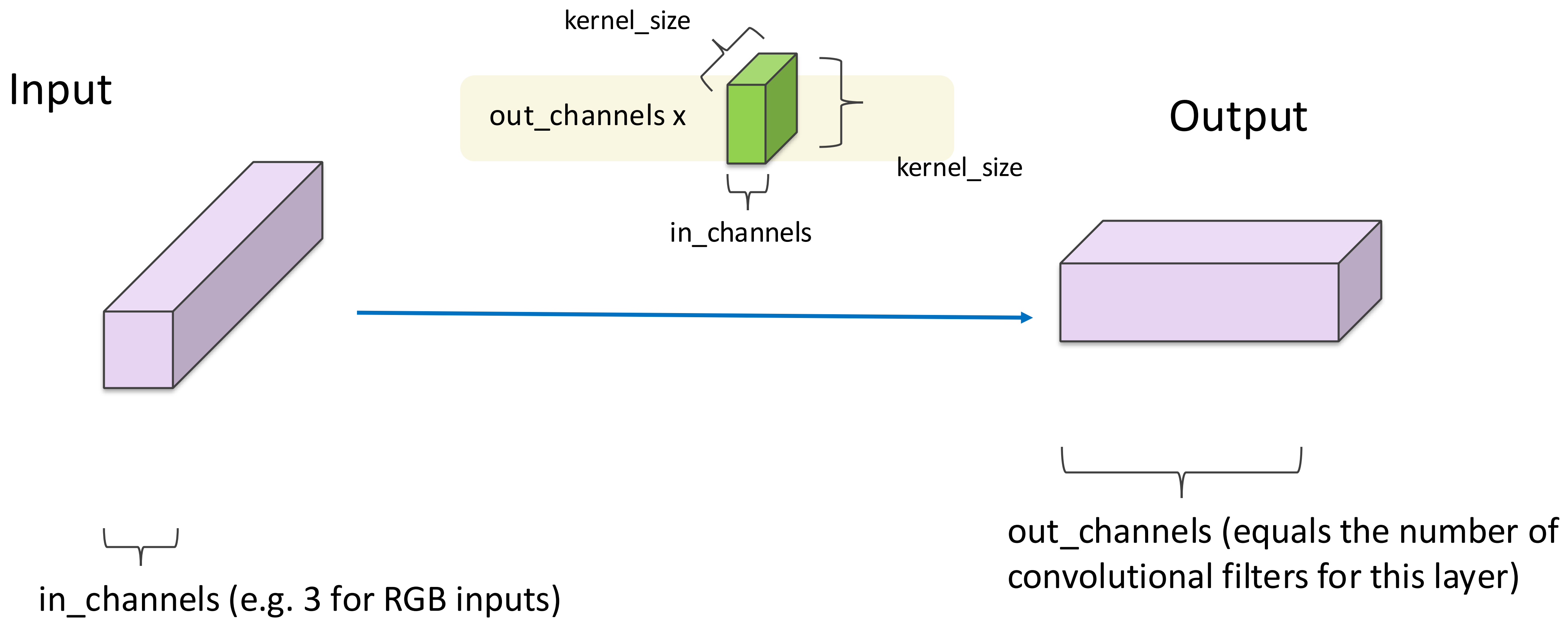
**Vladlen Koltun**
Intel Labs

ICLR 2016

# Convolutional Layer in pytorch



```
class torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)    [source]
```

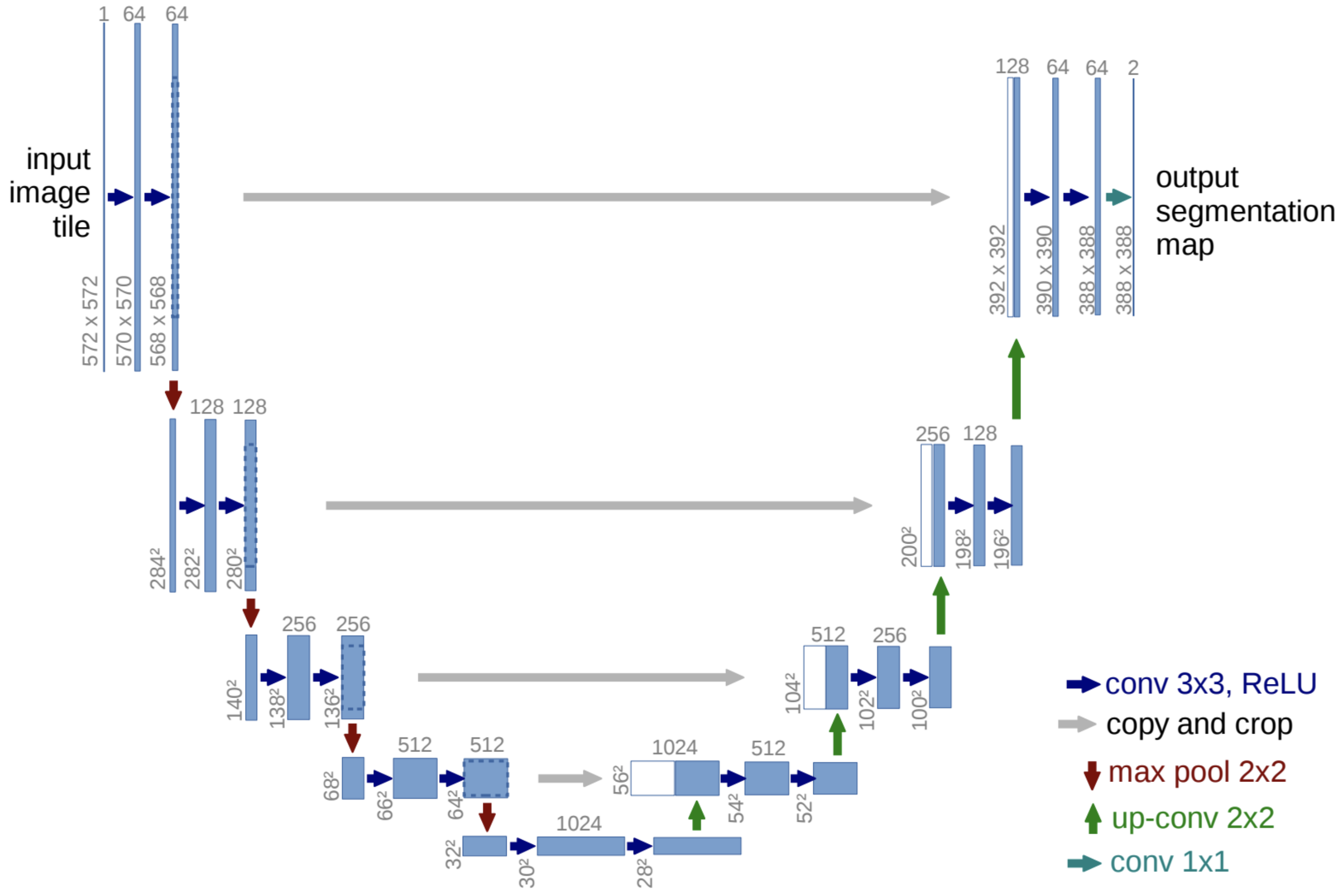


Input
kernel_size
out_channels x
kernel_size
in_channels
Output
in_channels (e.g. 3 for RGB inputs)
out_channels (equals the number of convolutional filters for this layer)

# U-Net: Convolutional Networks for Biomedical Image Segmentation

Olaf Ronneberger, Philipp Fischer, and Thomas Brox

Computer Science Department and BIOSS Centre for Biological Signalling Studies,
University of Freiburg, Germany

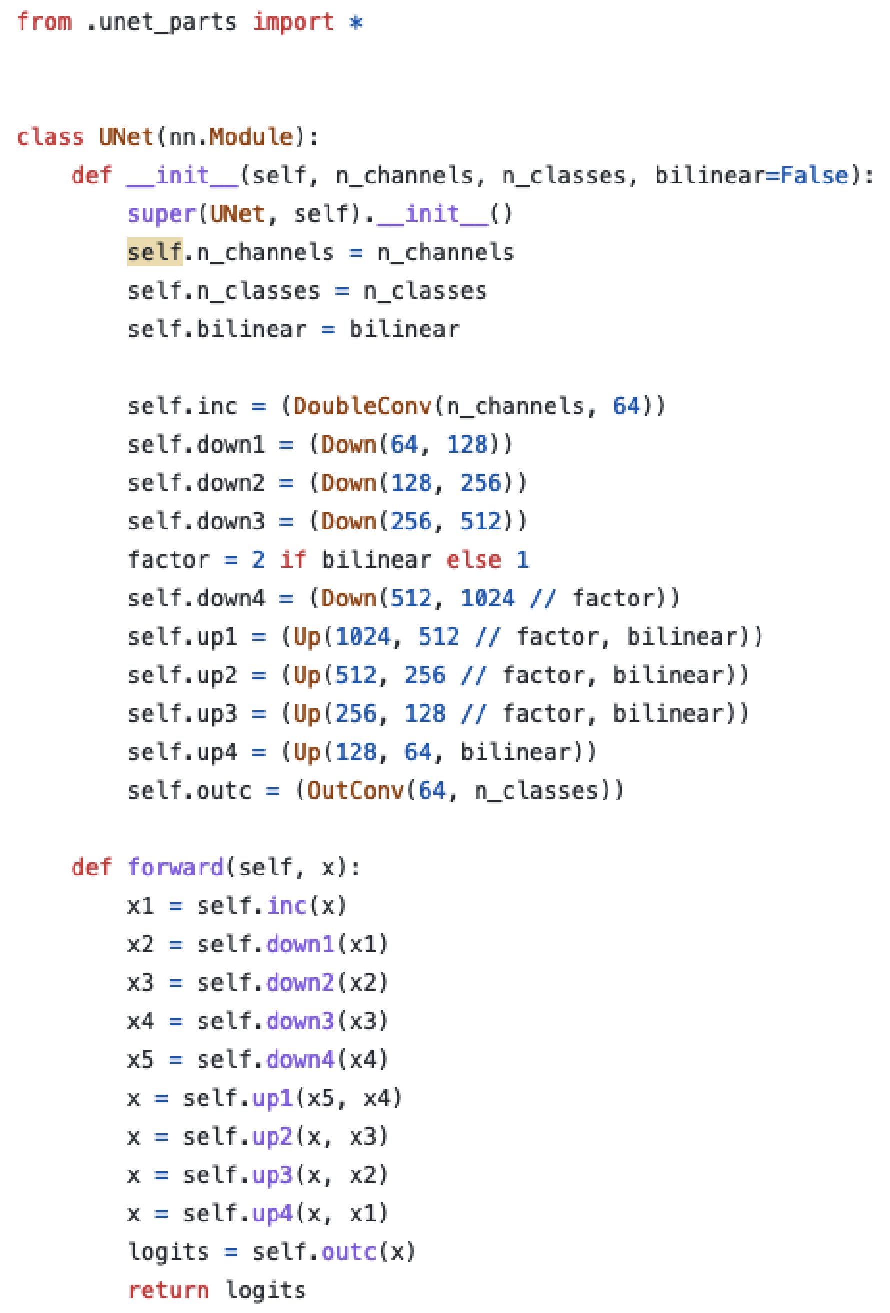


https://arxiv.org/abs/1505.04597

https://github.com/milesial/Pytorch-UNet

https://github.com/usuyama/pytorch-unet

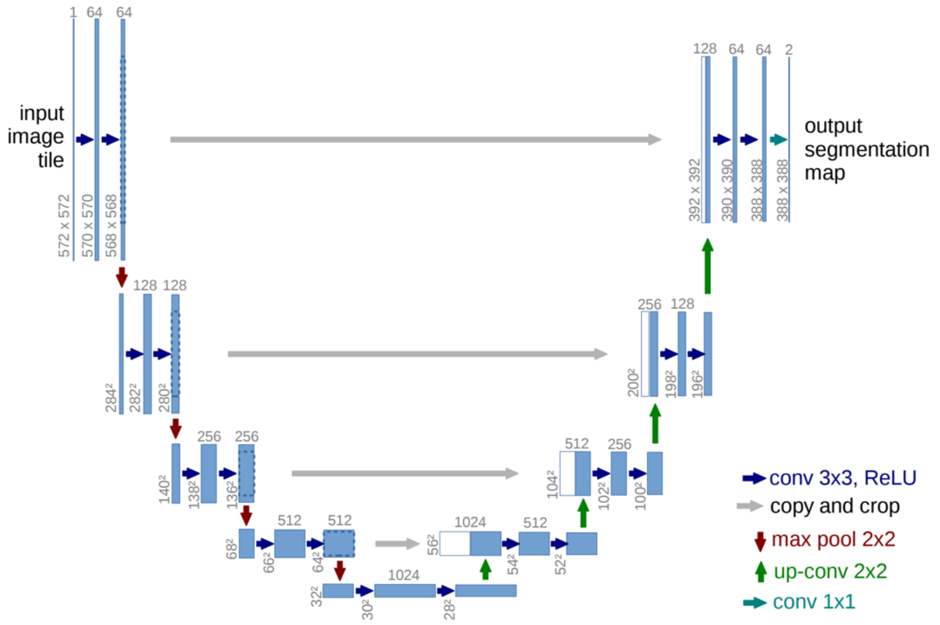
# UNet in Pytorch



```python
from .unet_parts import *


class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=False):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inc = (DoubleConv(n_channels, 64))
        self.down1 = (Down(64, 128))
        self.down2 = (Down(128, 256))
        self.down3 = (Down(256, 512))
        factor = 2 if bilinear else 1
        self.down4 = (Down(512, 1024 // factor))
        self.up1 = (Up(1024, 512 // factor, bilinear))
        self.up2 = (Up(512, 256 // factor, bilinear))
        self.up3 = (Up(256, 128 // factor, bilinear))
        self.up4 = (Up(128, 64, bilinear))
        self.outc = (OutConv(64, n_classes))

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        logits = self.outc(x)
        return logits
```

https://github.com/milesial/Pytorch-UNet/blob/master/unet/unet_model.py

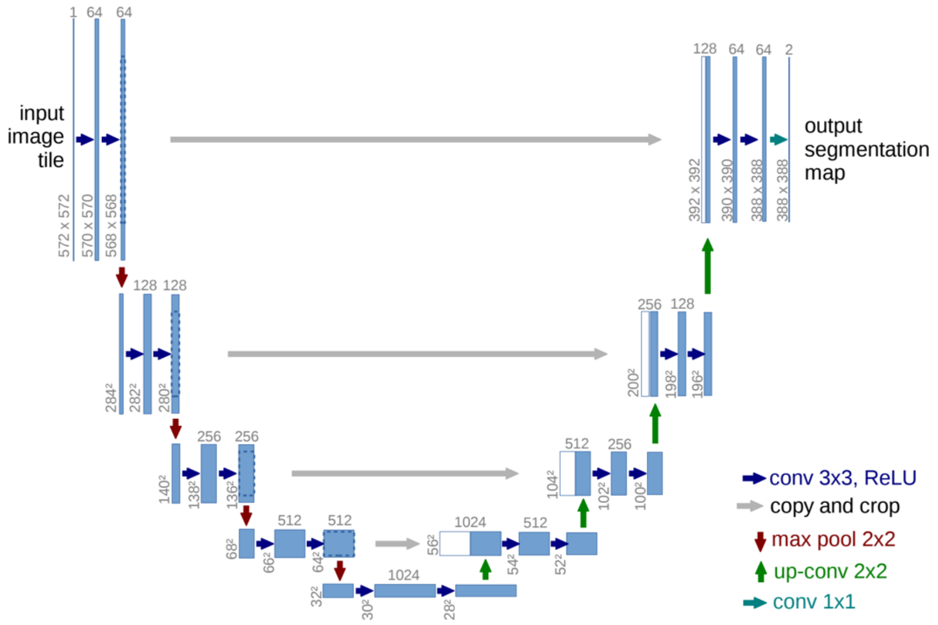
# Chair segmentation - Training

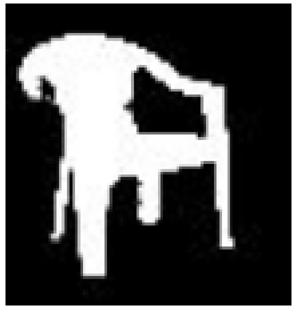


**Chair Segments: A Compact Benchmark for the Study of Object Segmentation**

Leticia Pinto-Alva[‡†], Ian K. Torres[♮*], Rosangel Garcia[§*], Ziyan Yang[†], Vicente Ordonez[†]
[‡]Universidad Católica San Pablo, [♮]University of Massachusetts, Amherst, [§]Le Moyne College, [†]University of Virginia
lp2rv@virginia.edu, zy3cx@virginia.edu, vicente@virginia.edu

# Chair segmentation - Prediction





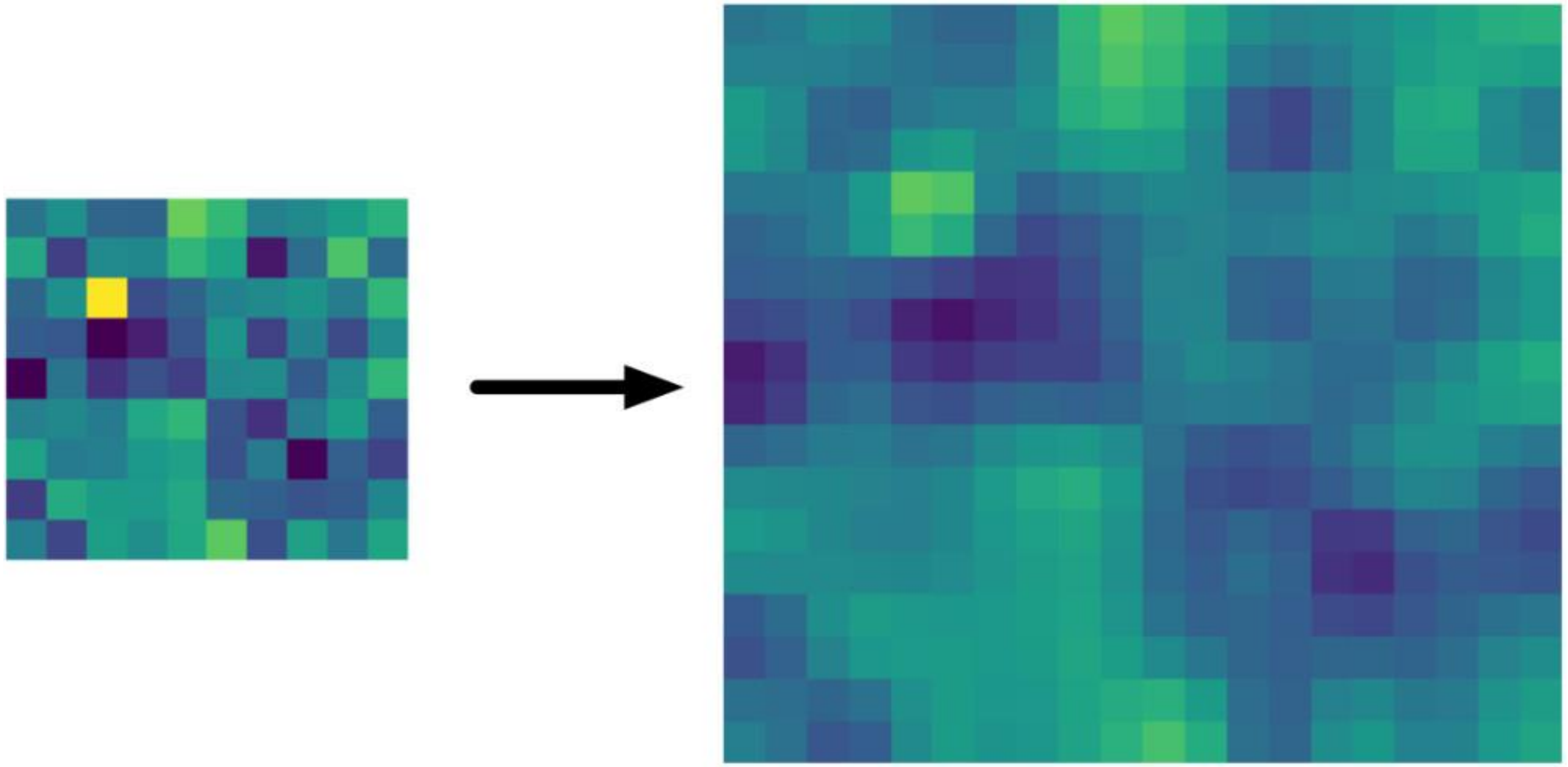
**Chair Segments: A Compact Benchmark for the Study of Object Segmentation**

Leticia Pinto-Alva[‡†], Ian K. Torres[♮*], Rosangel Garcia[§*], Ziyan Yang[†], Vicente Ordonez[†]

[‡]Universidad Católica San Pablo, [♮]University of Massachusetts, Amherst, [§]Le Moyne College,
[†]University of Virginia

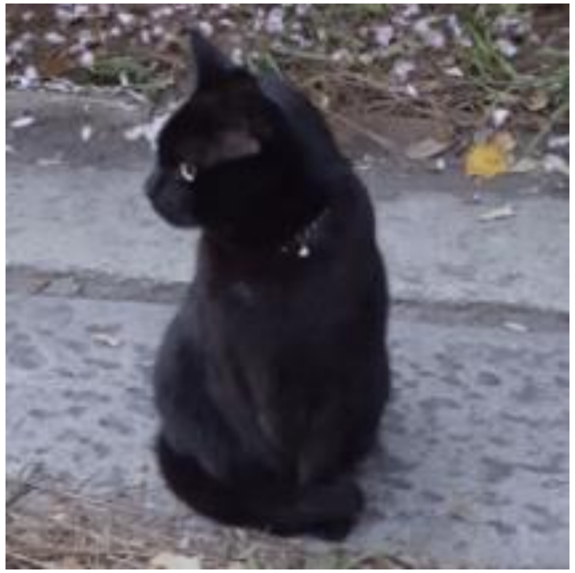
lp2rv@virginia.edu, zy3cx@virginia.edu, vicente@virginia.edu

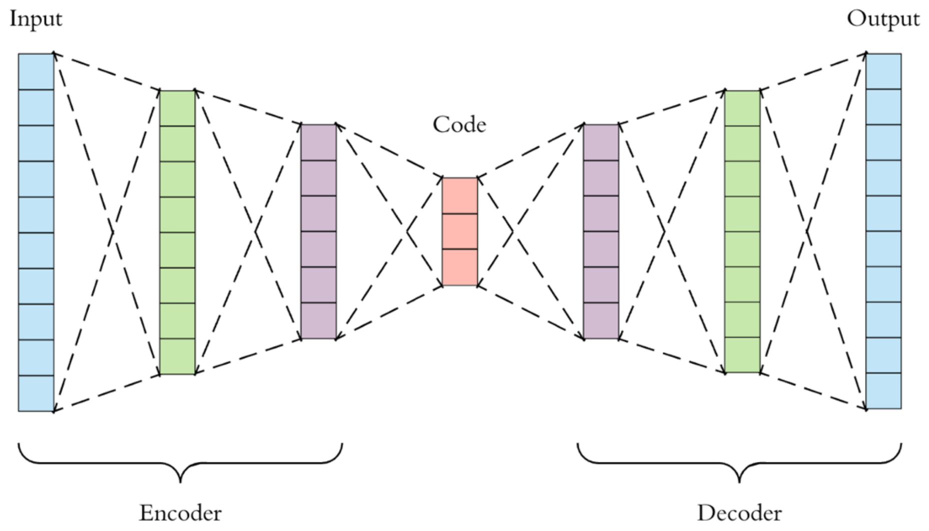
# Bilinear Upsampling Layer

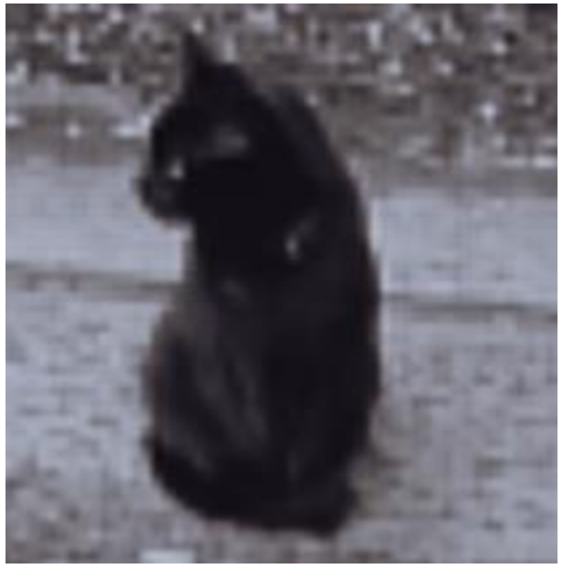


https://machinethink.net/blog/coreml-upsampling/

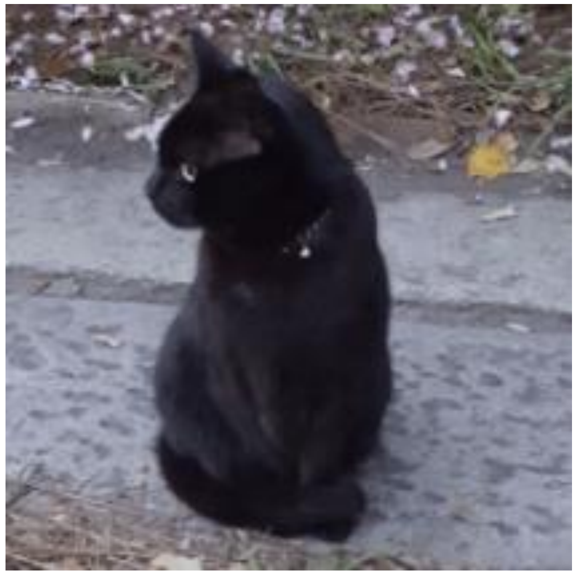
# AutoEncoder Models (Downsample, Upsample)

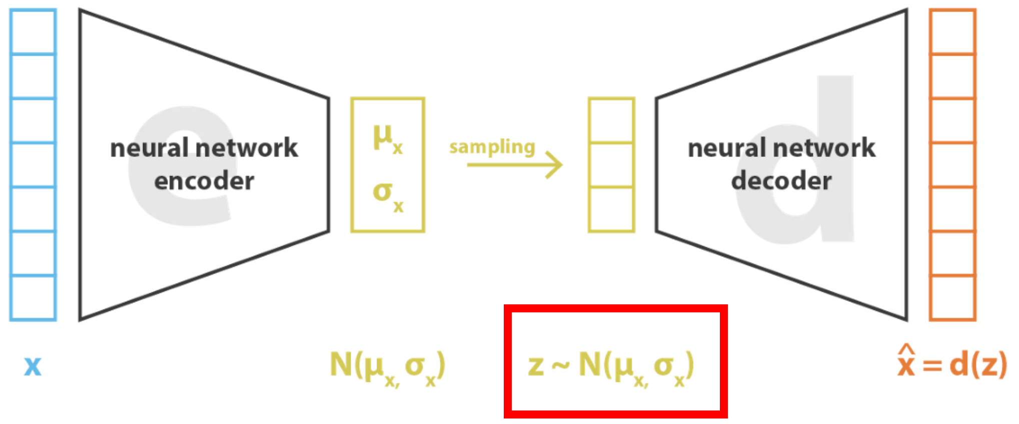






# Variational AutoEncoders (VAE)





$$\text{loss} = \|x - \hat{x}\|^2 + \mathrm{KL}[\,N(\mu_x, \sigma_x), N(0, I)\,]$$

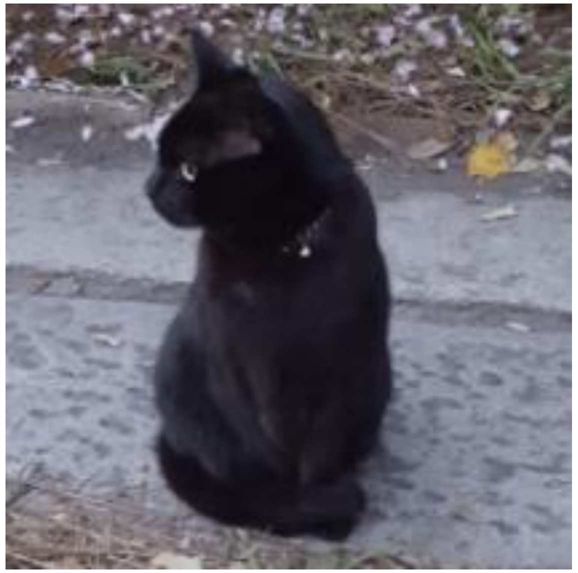
https://pytorch.org/docs/stable/generated/torch.normal.html
https://ai.stackexchange.com/questions/30176/are-mean-and-standard-deviation-in-variational-autoencoders-unique

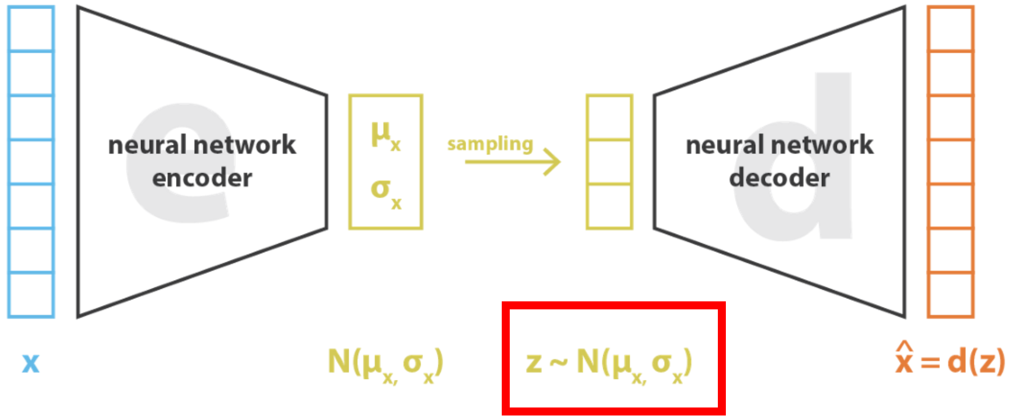
# Reparameterization "trick"

$$z = z\_mean + sigma * epsilon$$

$$sigma = exp(z\_log\_var/2)$$

$$\varepsilon \sim \text{Normal}(0,1)$$



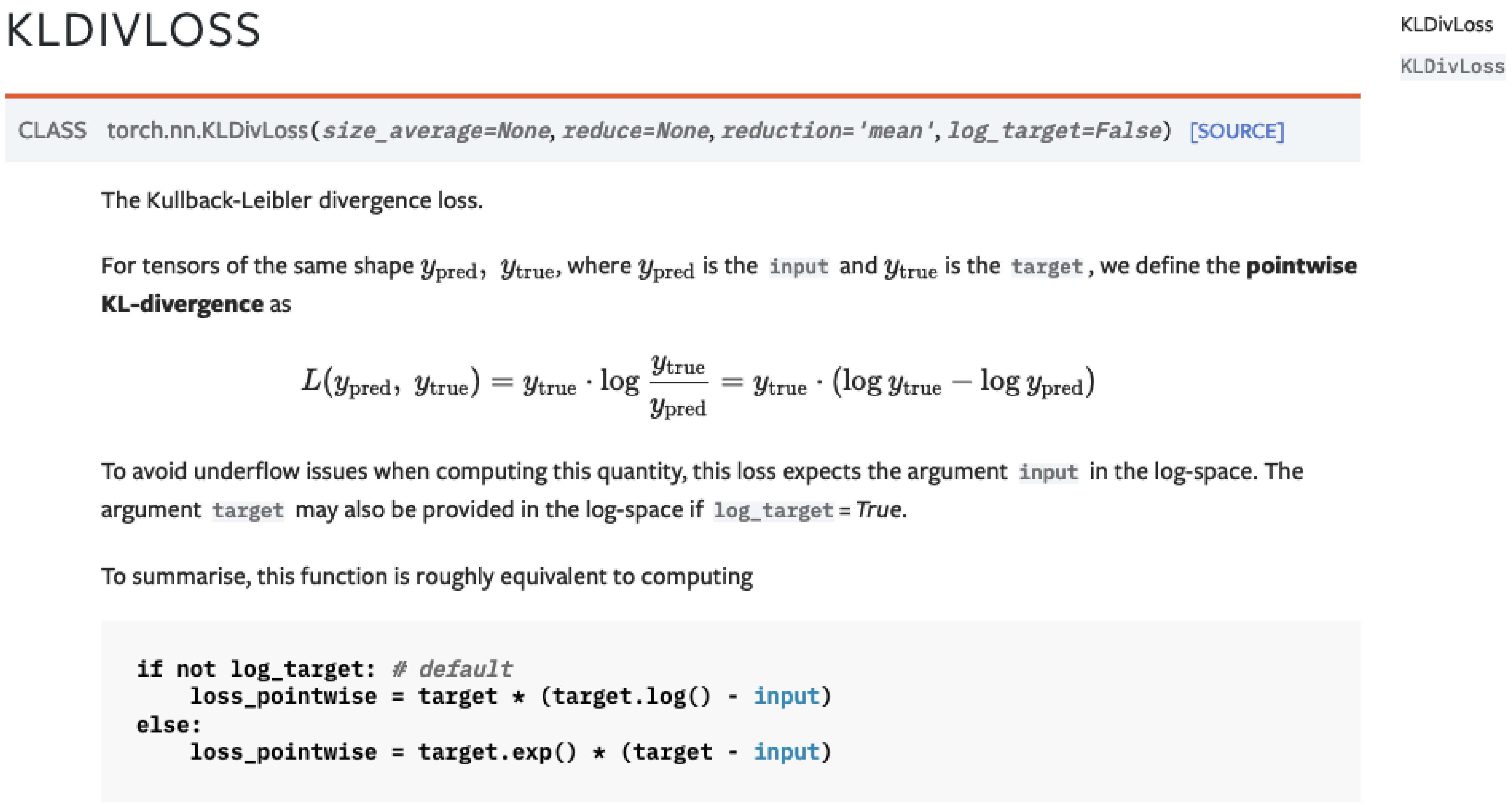


$$\text{loss} = \|x - \hat{x}\|^2 + \text{KL}[N(\mu_x, \sigma_x), N(0, I)]$$

https://pytorch.org/docs/stable/generated/torch.normal.html
https://ai.stackexchange.com/questions/30176/are-mean-and-standard-deviation-in-variational-autoencoders-unique

## Kullback-Leibler Divergence

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \, \log\left( \frac{P(x)}{Q(x)} \right)$$



# KLDIVLOSS

KLDivLoss

`KLDivLoss`

CLASS torch.nn.KLDivLoss(*size_average=None*, *reduce=None*, *reduction='mean'*, *log_target=False*) [SOURCE]

The Kullback-Leibler divergence loss.

For tensors of the same shape $y_{\text{pred}},\ y_{\text{true}}$, where $y_{\text{pred}}$ is the `input` and $y_{\text{true}}$ is the `target`, we define the **pointwise KL-divergence** as

$$L(y_{\text{pred}},\ y_{\text{true}}) = y_{\text{true}} \cdot \log \frac{y_{\text{true}}}{y_{\text{pred}}} = y_{\text{true}} \cdot (\log y_{\text{true}} - \log y_{\text{pred}})$$

To avoid underflow issues when computing this quantity, this loss expects the argument `input` in the log-space. The argument `target` may also be provided in the log-space if `log_target` = *True*.

To summarise, this function is roughly equivalent to computing

```
if not log_target: # default
    loss_pointwise = target * (target.log() - input)
else:
    loss_pointwise = target.exp() * (target - input)
```

# For Gaussian Probabilities KL Divergence even simpler

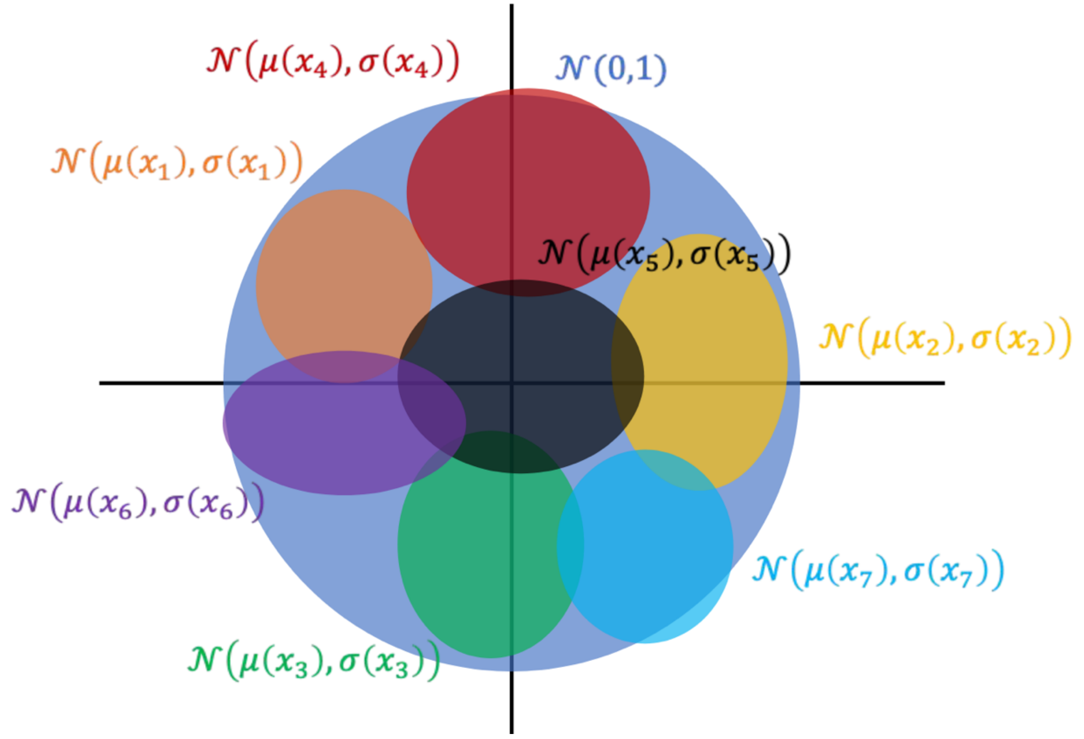
$$D_{KL}(\mathcal{N}(\mu,\sigma^2)||\mathcal{N}(0,1)) = \frac{1}{2}\sum_{i=1}^{k}(\mu_i^2+\sigma_i^2-\ln\sigma_i^2-1)$$

```
def loss_function(self,
                  *args,
                  **kwargs) -> dict:
    recons = args[0]
    input = args[1]
    mu = args[2]
    log_var = args[3]

    kld_weight = kwargs['M_N']  # Account for the minibatch samples from the dataset
    recons_loss =F.mse_loss(recons, input)

    kld_loss = torch.mean(-0.5 * torch.sum(1 + log_var - mu ** 2 - log_var.exp(), dim = 1), dim = 0)

    loss = recons_loss + kld_weight * kld_loss
    return {'loss': loss, 'Reconstruction_Loss':recons_loss, 'KLD':-kld_loss}
```

https://github.com/AntixK/PyTorch-VAE/blob/master/models/cvae.py

# Encoding different points into latent space

$$L = ||x - \hat{x}||_2^2 + \lambda D_{KL}(\mathcal{N}(\mu, \sigma), \mathcal{N}(0, 1))$$



$\mathcal{N}(\mu(x_4), \sigma(x_4))$
$\mathcal{N}(0,1)$
$\mathcal{N}(\mu(x_1), \sigma(x_1))$
$\mathcal{N}(\mu(x_5), \sigma(x_5))$
$\mathcal{N}(\mu(x_2), \sigma(x_2))$
$\mathcal{N}(\mu(x_6), \sigma(x_6))$
$\mathcal{N}(\mu(x_7), \sigma(x_7))$
$\mathcal{N}(\mu(x_3), \sigma(x_3))$

Because of our KL divergence loss, the $\mathcal{N}(\mu, \sigma)$ for any input data point has to be somewhat similar to $\mathcal{N}(0,1)$

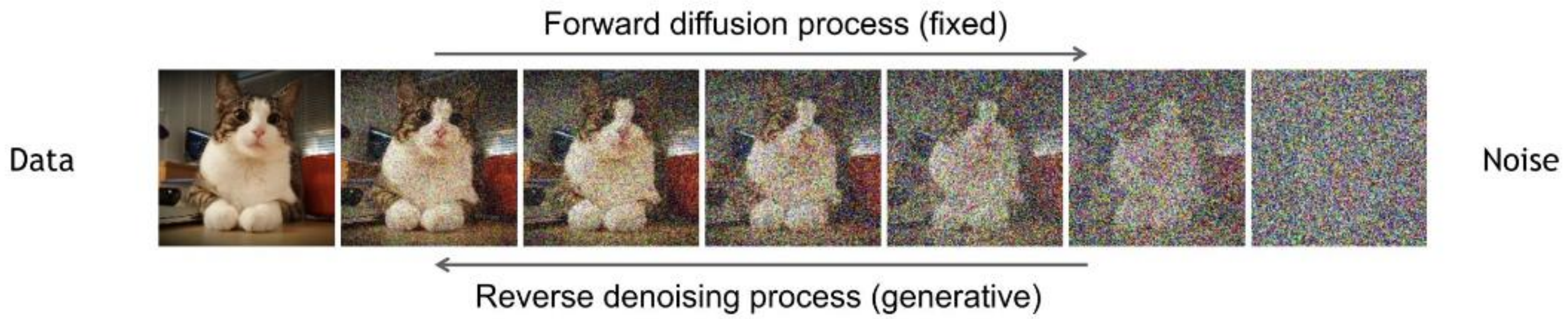
So, if we sample a point from $\mathcal{N}(0,1)$, it is very likely to fall within one of these encoded

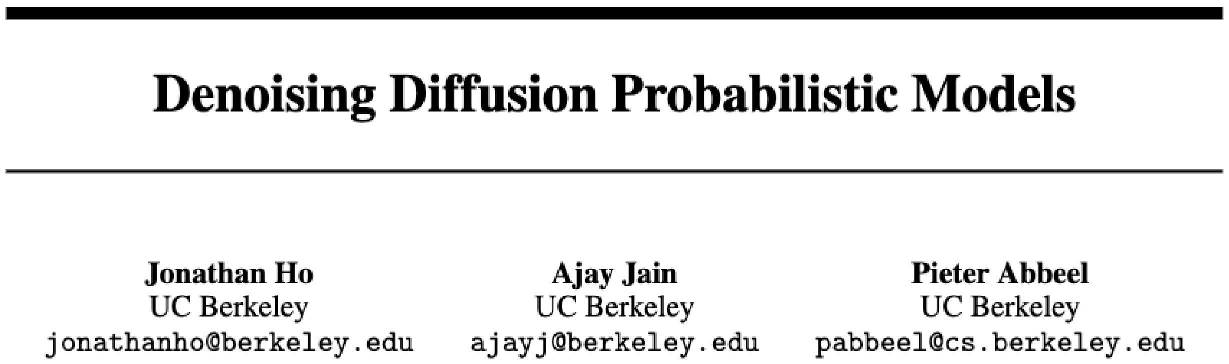
Slide from Ritambara Singh's Deep Learning class at Brown
https://brown-deep-learning.github.io/dl-website-s23/slides/lecture27.pdf

## Denoising Diffusion Probabilistic Models (DDPM)

**Forward diffusion:** Markov chain of diffusion steps to slowly add gaussian noise to data

**Reverse diffusion:** A model is trained to generate data from noise by iterative denoising


Forward diffusion process (fixed)
Data
Noise
Reverse denoising process (generative)


**Denoising Diffusion Probabilistic Models**

**Jonathan Ho**
UC Berkeley
jonathanho@berkeley.edu

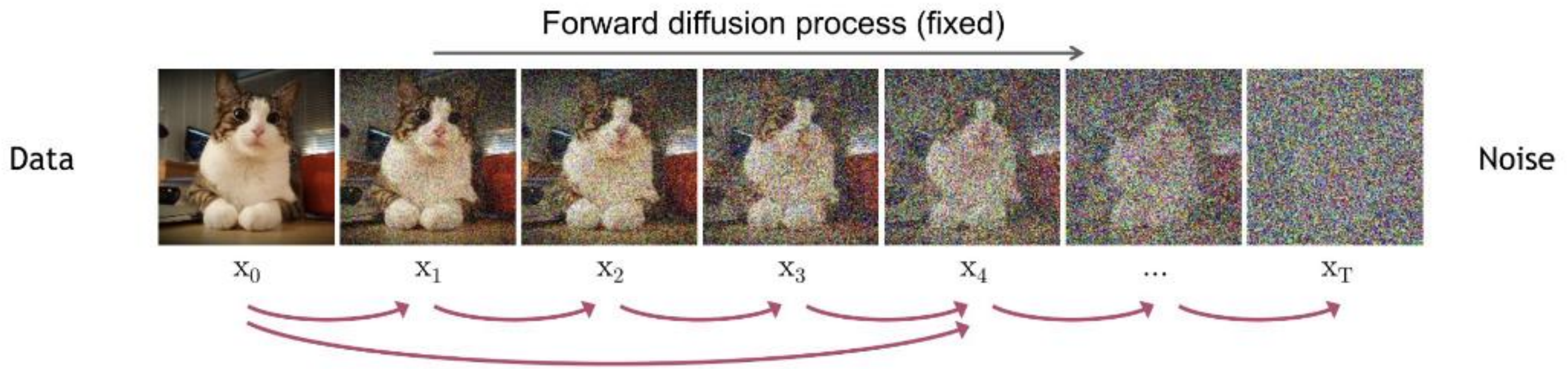
**Ajay Jain**
UC Berkeley
ajayj@berkeley.edu

**Pieter Abbeel**
UC Berkeley
pabbeel@cs.berkeley.edu

https://cvpr2022-tutorial-diffusion-models.github.io/

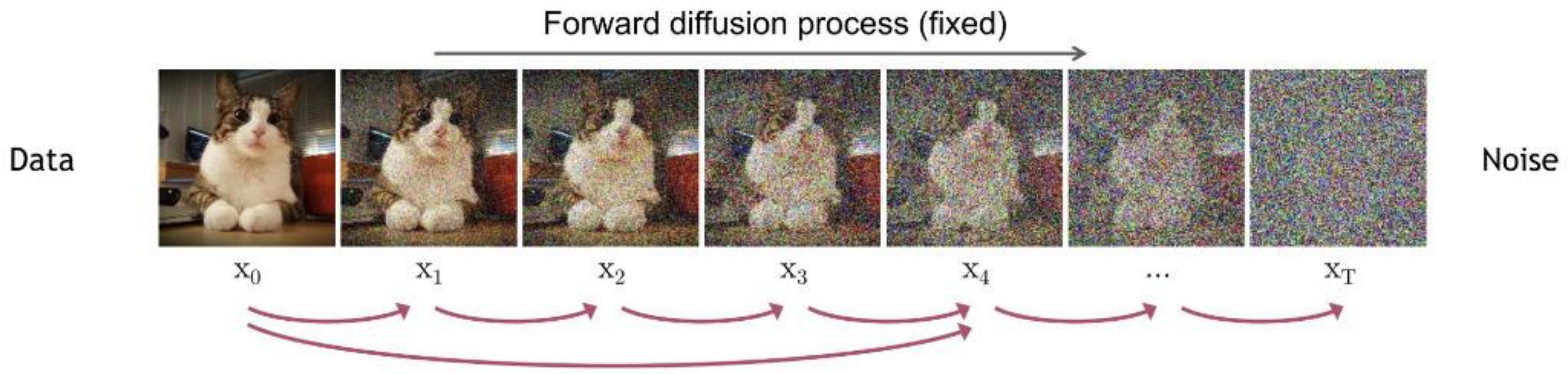
Slides compiled by my student Aman Shrivastava

## DDPM | Forward diffusion



We add a small amount of gaussian noise to a sample $\mathbf{x}_0$ in **T** timesteps to produces noised samples, $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T\}$. The steps are controlled by the noise schedule as follows:

$$\boxed{q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \quad q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})}$$

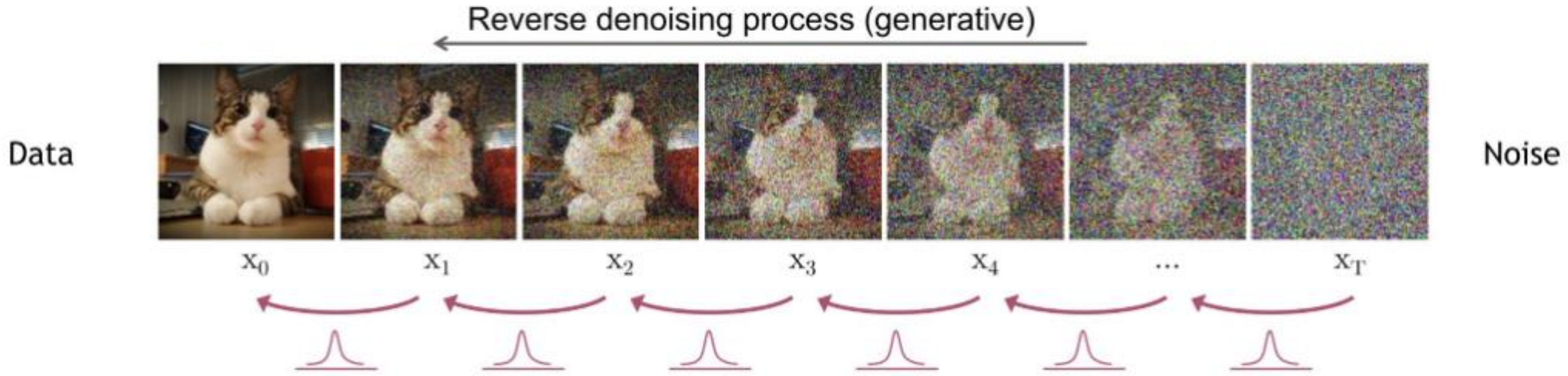


$$\boxed{q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \quad q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})}$$

Define $\bar{\alpha}_t = \prod_{s=1}^{t}(1-\beta_s)$ ➡ $q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}))$ (Diffusion Kernel)

For sampling: $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\ \mathbf{x}_0 + \sqrt{(1-\bar{\alpha}_t)}\ \epsilon$ where $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$

## DDPM | Reverse Diffusion

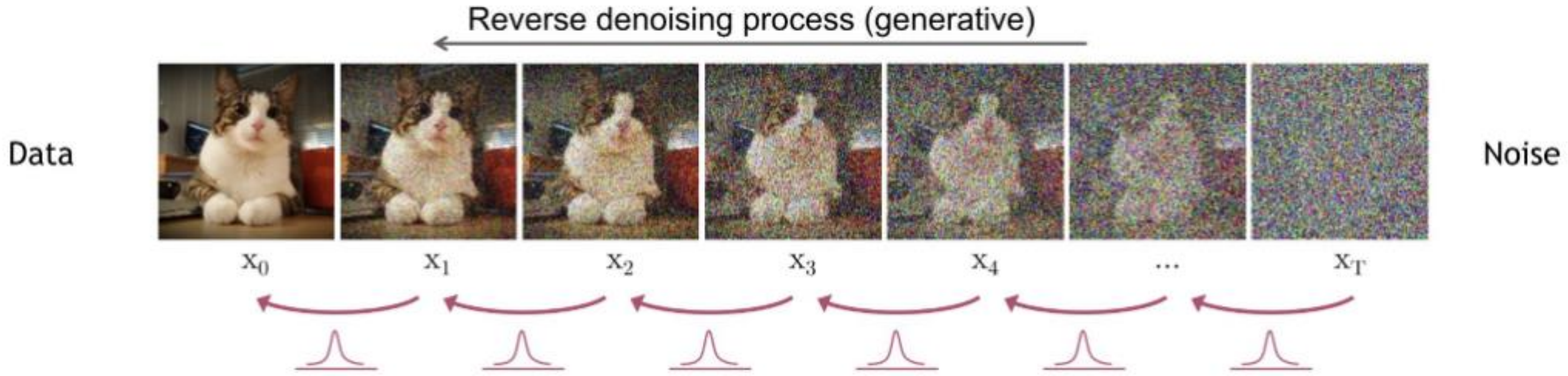


We learn a neural network model **($p_\theta$)** to approximate these conditional probabilities **$q(x_{(t-1)} \mid x_t)$** in order to run the reverse diffusion process as follows:

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \quad p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$$

https://cvpr2022-tutorial-diffusion-models.github.io/

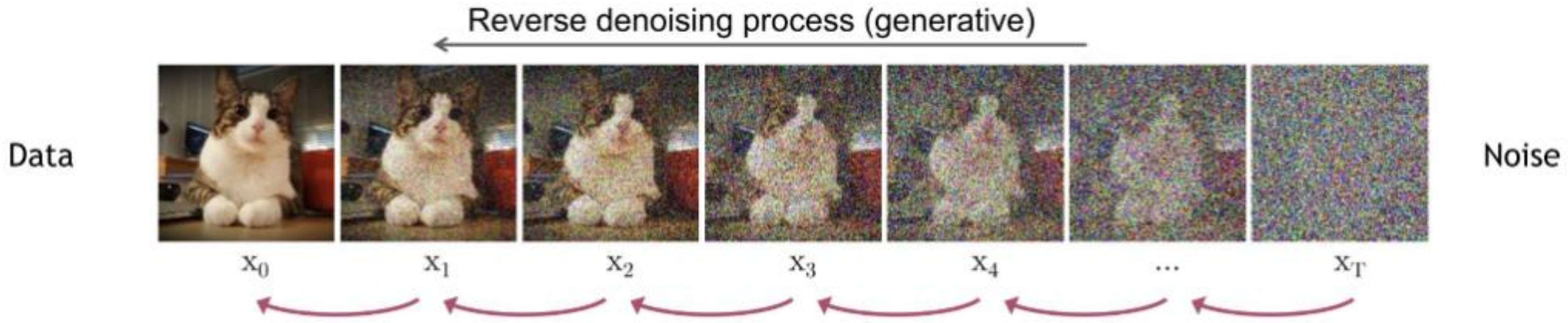
## DDPM | Reverse Diffusion



We learn a neural network model **($p_\theta$)** to approximate these conditional probabilities **$q(x_{(t-1)} \mid x_t)$** in order to run the reverse diffusion process as follows:

$$\boxed{p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T)\prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \quad p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))}$$

# How do we train?



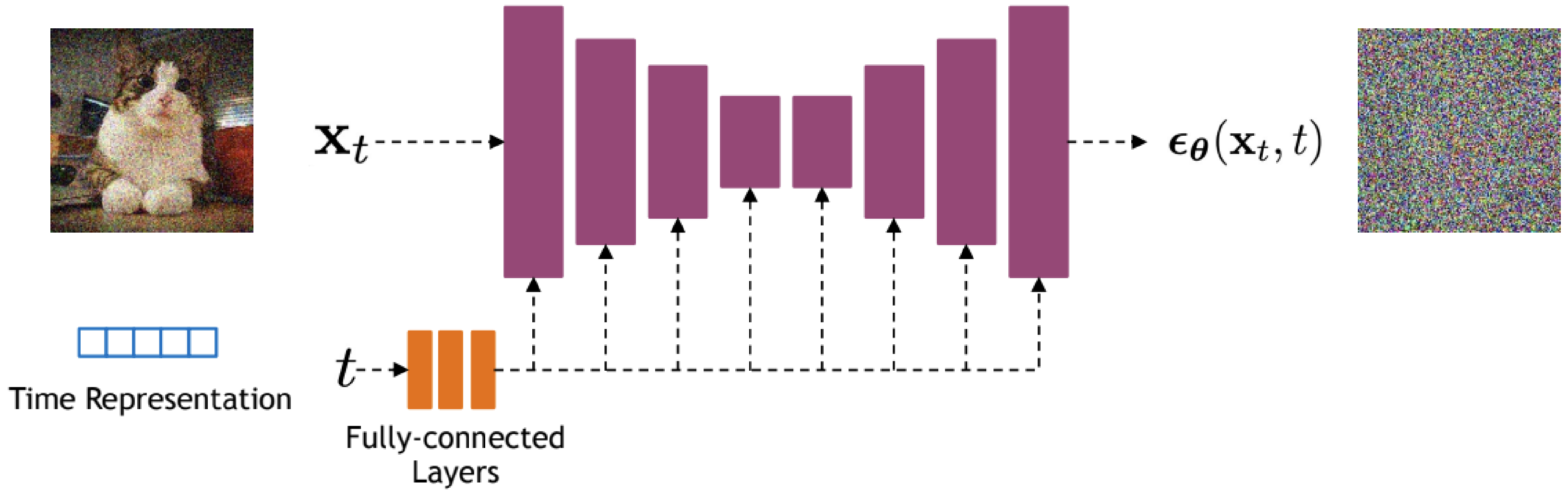
**Algorithm 1** Training

```
1: repeat
2:   $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3:   $t \sim \mathrm{Uniform}(\{1, \ldots, T\})$
4:   $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5:   Take gradient descent step on
         $\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$
6: until converged
```

https://cvpr2022-tutorial-diffusion-models.github.io/

# Unet to model transition

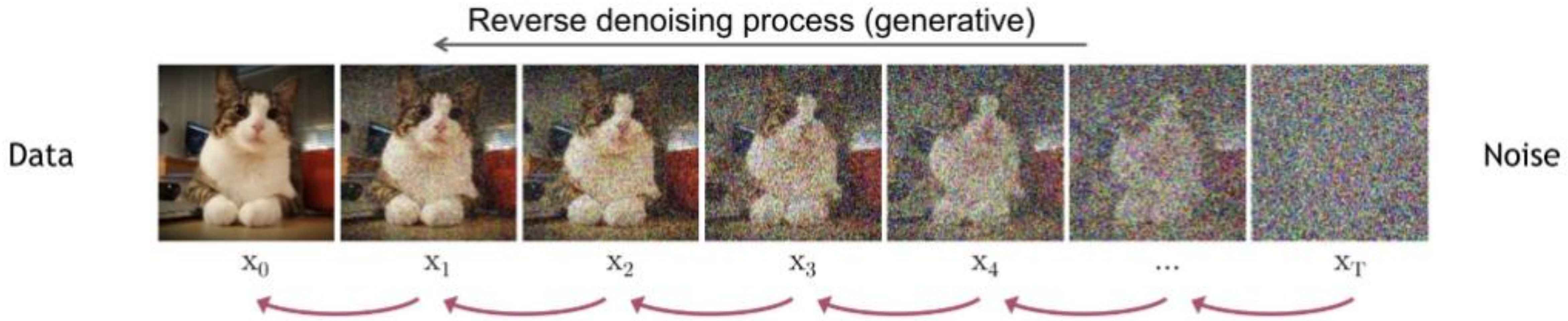
Diffusion models often use U-Net architectures with ResNet blocks and self-attention layers to represent $\epsilon_\theta(\mathbf{x}_t, t)$

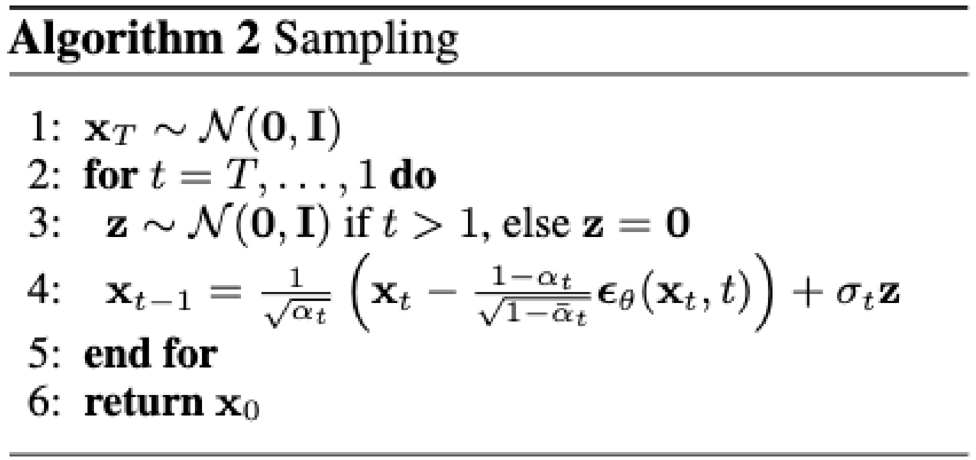


Time representation: sinusoidal positional embeddings or random Fourier features.

Time features are fed to the residual blocks using either simple spatial addition or using adaptive group normalization layers. (see Dharivwal and Nichol NeurIPS 2021)

# How do we train?



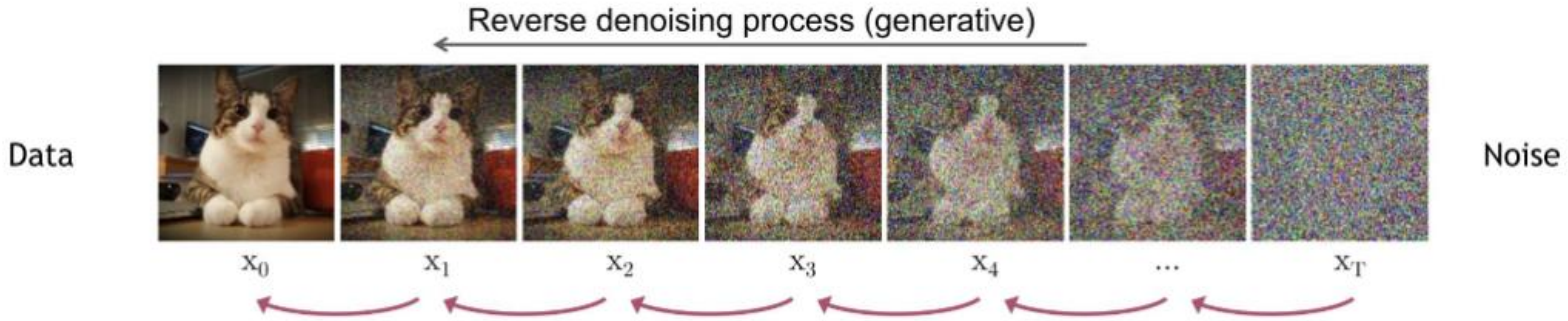


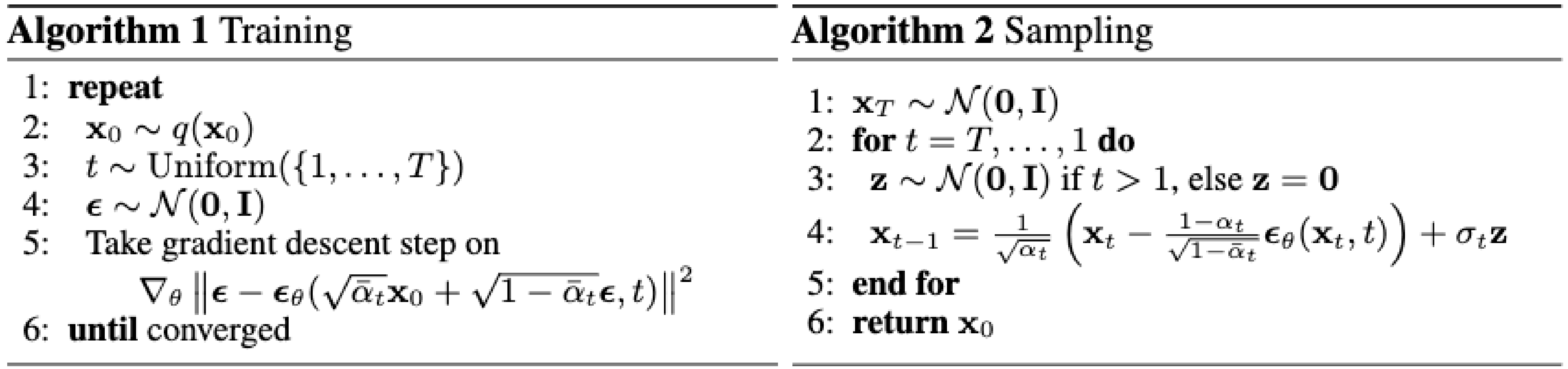
**Algorithm 2** Sampling

```
1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: for $t = T, \dots, 1$ do
3:   $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4:   $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$
5: end for
6: return $\mathbf{x}_0$
```

https://cvpr2022-tutorial-diffusion-models.github.io/

# How do we train?





**Algorithm 1** Training

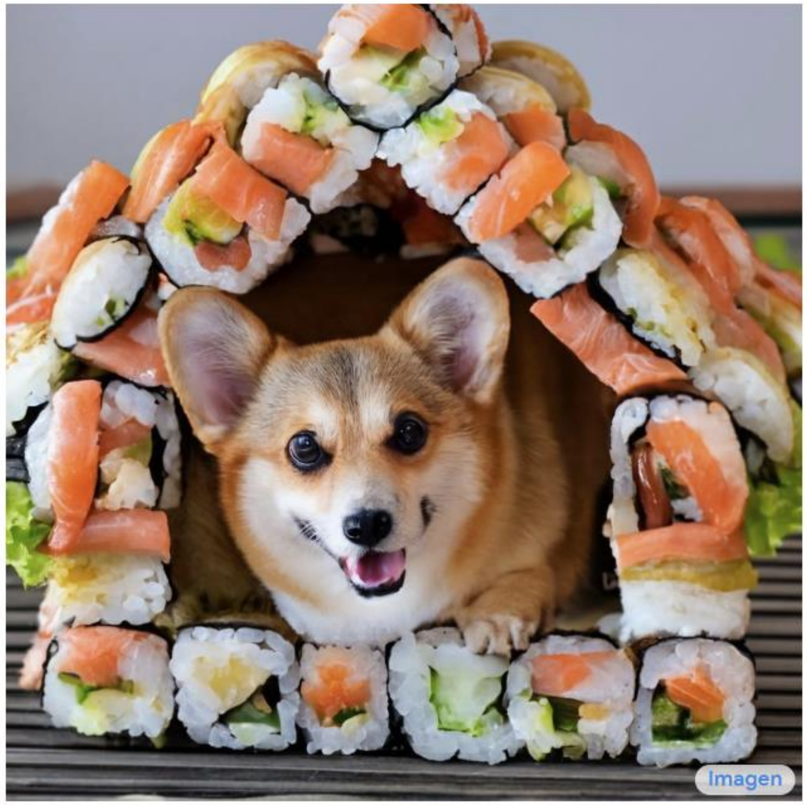
```
1: repeat
2:   $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3:   $t \sim \mathrm{Uniform}(\{1, \ldots, T\})$
4:   $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5:   Take gradient descent step on
         $\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$
6: until converged
```

**Algorithm 2** Sampling

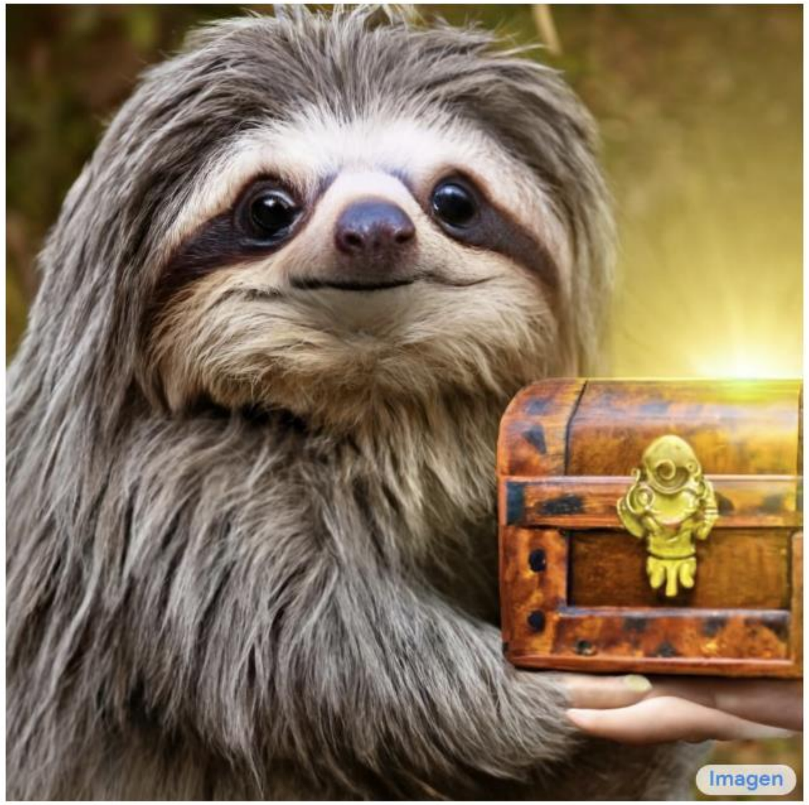
```
1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: for $t = T, \ldots, 1$ do
3:   $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4:   $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left( \mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: end for
6: return $\mathbf{x}_0$
```

https://cvpr2022-tutorial-diffusion-models.github.io/

# Imagen by Google

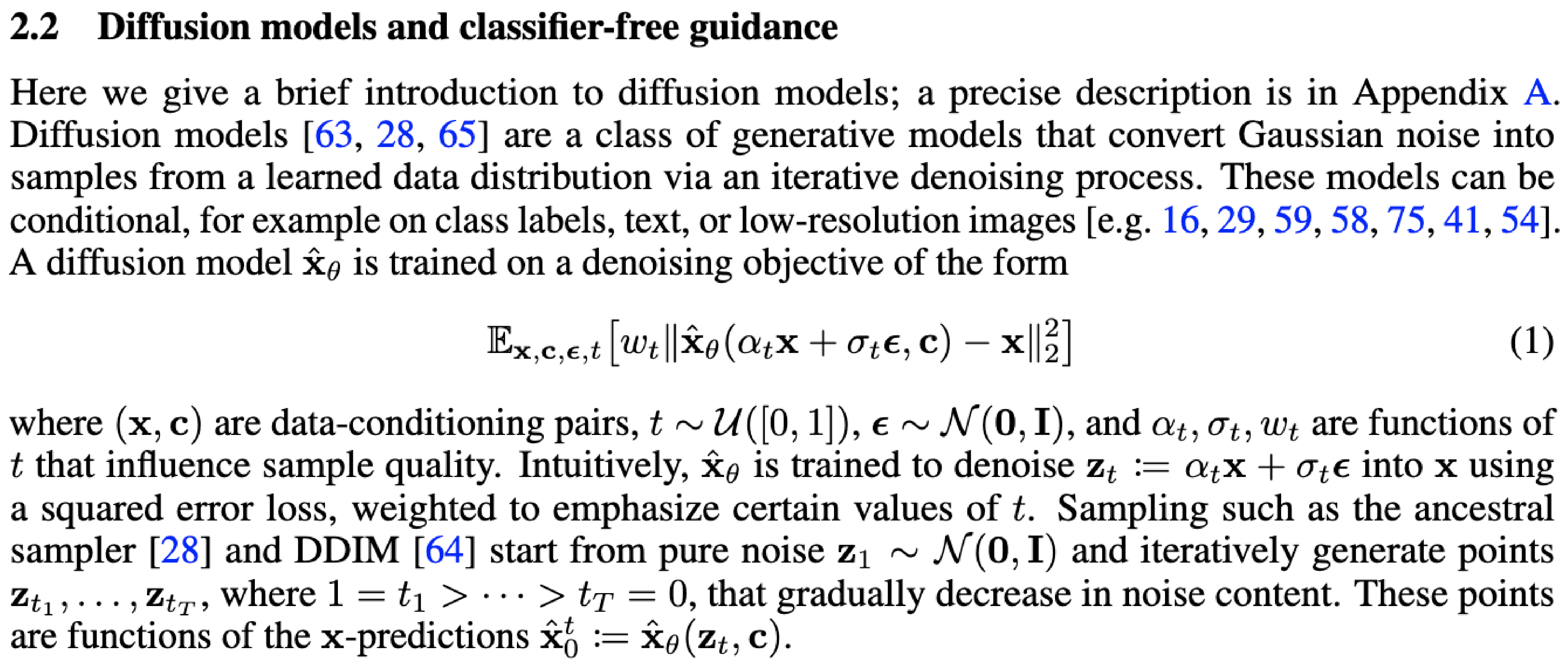


A cute corgi lives in a house made out of sushi.



A cute sloth holding a small treasure chest. A bright golden glow is coming from the chest.

# Imagen by Google



## 2.2 Diffusion models and classifier-free guidance

Here we give a brief introduction to diffusion models; a precise description is in Appendix A. Diffusion models [63, 28, 65] are a class of generative models that convert Gaussian noise into samples from a learned data distribution via an iterative denoising process. These models can be conditional, for example on class labels, text, or low-resolution images [e.g. 16, 29, 59, 58, 75, 41, 54]. A diffusion model $\hat{\mathbf{x}}_\theta$ is trained on a denoising objective of the form

$$\mathbb{E}_{\mathbf{x},\mathbf{c},\boldsymbol{\epsilon},t}\left[w_t\|\hat{\mathbf{x}}_\theta(\alpha_t\mathbf{x}+\sigma_t\boldsymbol{\epsilon},\mathbf{c})-\mathbf{x}\|_2^2\right] \tag{1}$$

where $(\mathbf{x},\mathbf{c})$ are data-conditioning pairs, $t\sim\mathcal{U}([0,1])$, $\boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I})$, and $\alpha_t,\sigma_t,w_t$ are functions of $t$ that influence sample quality. Intuitively, $\hat{\mathbf{x}}_\theta$ is trained to denoise $\mathbf{z}_t:=\alpha_t\mathbf{x}+\sigma_t\boldsymbol{\epsilon}$ into $\mathbf{x}$ using a squared error loss, weighted to emphasize certain values of $t$. Sampling such as the ancestral sampler [28] and DDIM [64] start from pure noise $\mathbf{z}_1\sim\mathcal{N}(\mathbf{0},\mathbf{I})$ and iteratively generate points $\mathbf{z}_{t_1},\dots,\mathbf{z}_{t_T}$, where $1=t_1>\cdots>t_T=0$, that gradually decrease in noise content. These points are functions of the $\mathbf{x}$-predictions $\hat{\mathbf{x}}_0^t:=\hat{\mathbf{x}}_\theta(\mathbf{z}_t,\mathbf{c})$.

https://arxiv.org/pdf/2205.11487.pdf

## Questions