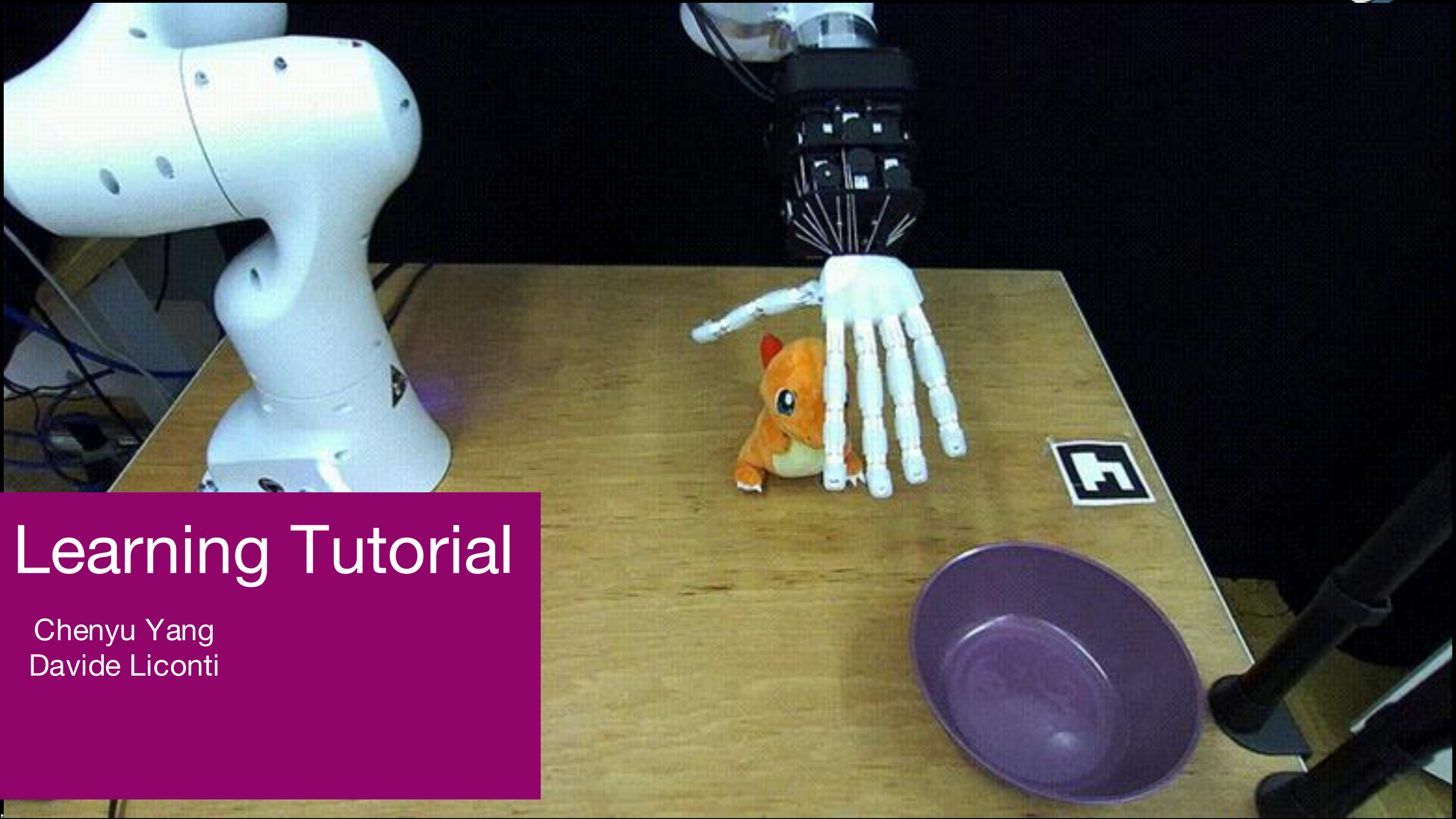
ETH zürich

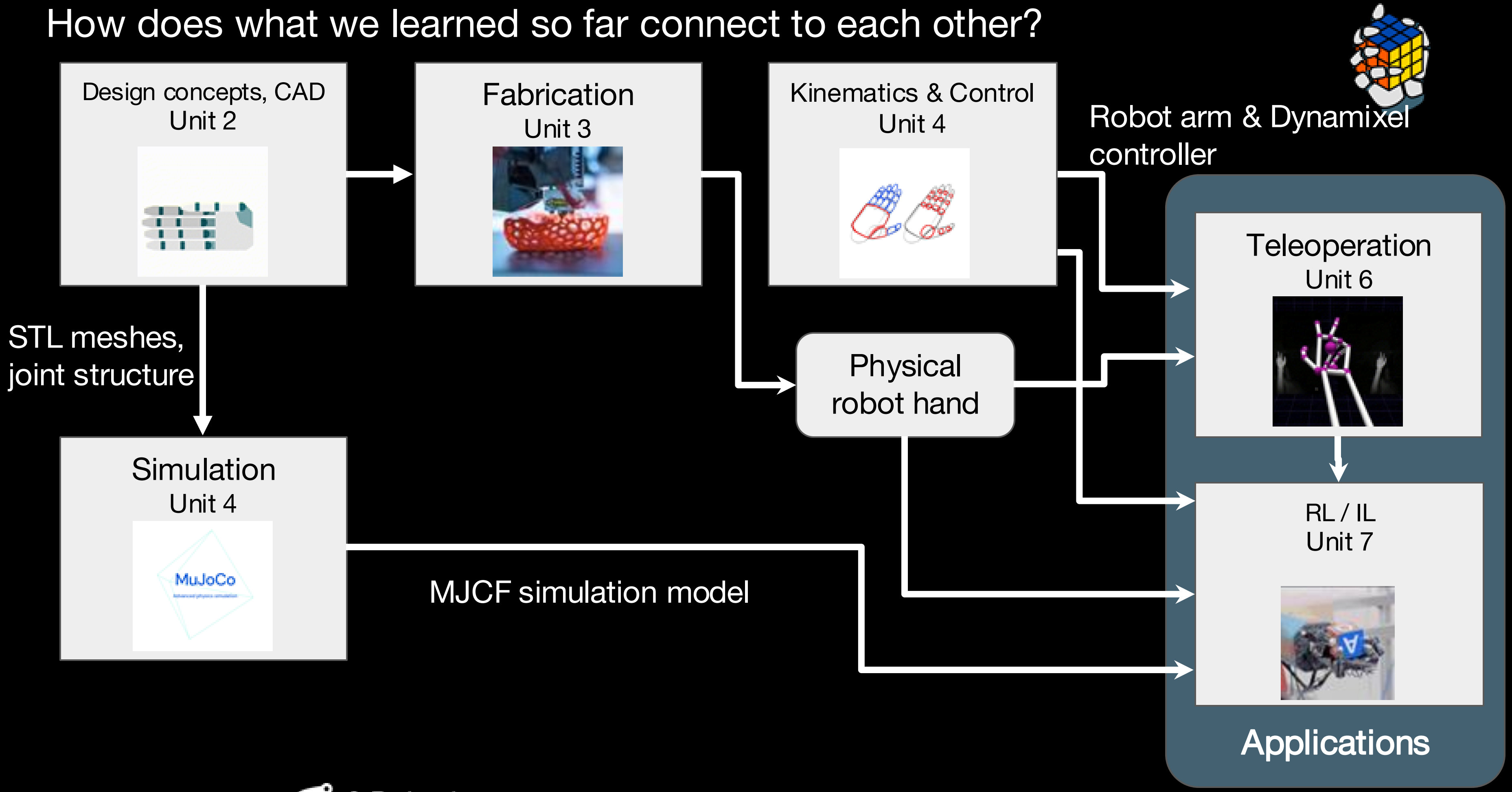


Imitation Learning Tutorial

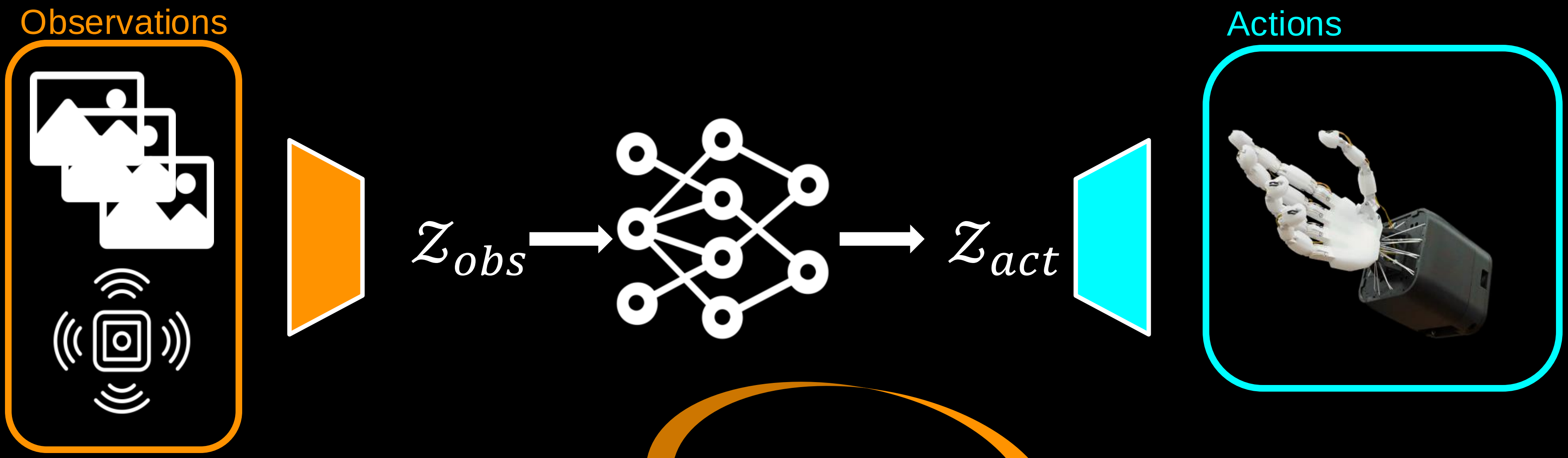
Chenyu Yang
Davide Liconti

SoftRobotics Laboratory

How does what we learned so far connect to each other?



Overview



What observations can we use?

What's the best z_{obs}?

What's the best z_{act}?

What's the best way to learn?

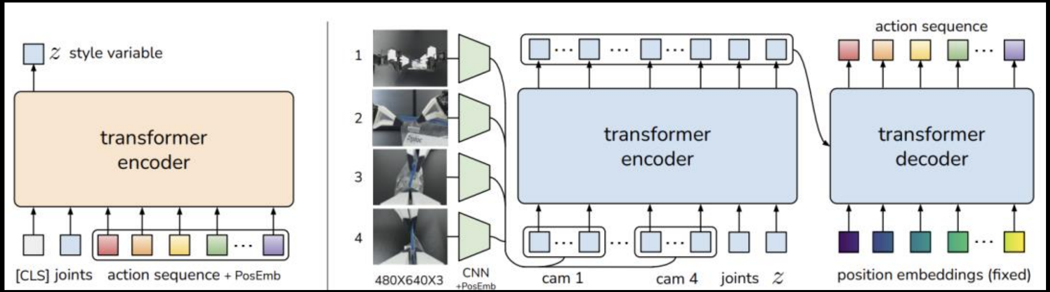
For robotic hands !

Contents

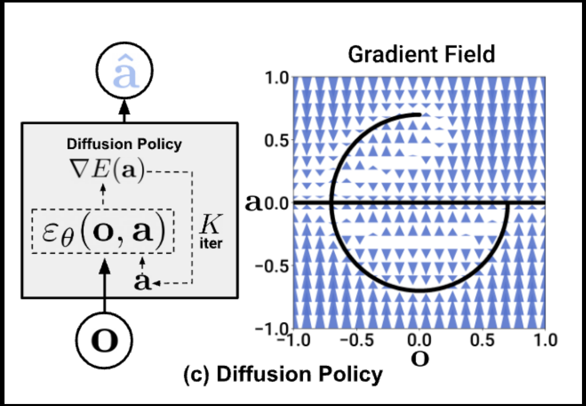
2 architectures for downstream-task imitation learning for robotics
ACT (paper)
Diffusion Policy(paper)

What we need to understand today:

1. Transformer Architecture
2. Variational Autoencoder
3. Diffusion Mechanism

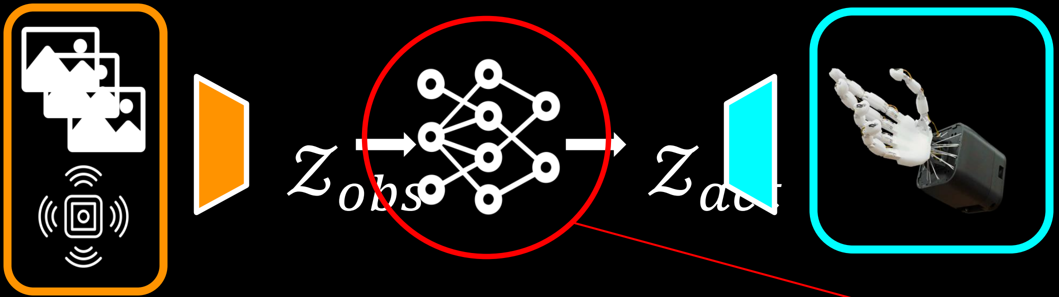


ACT



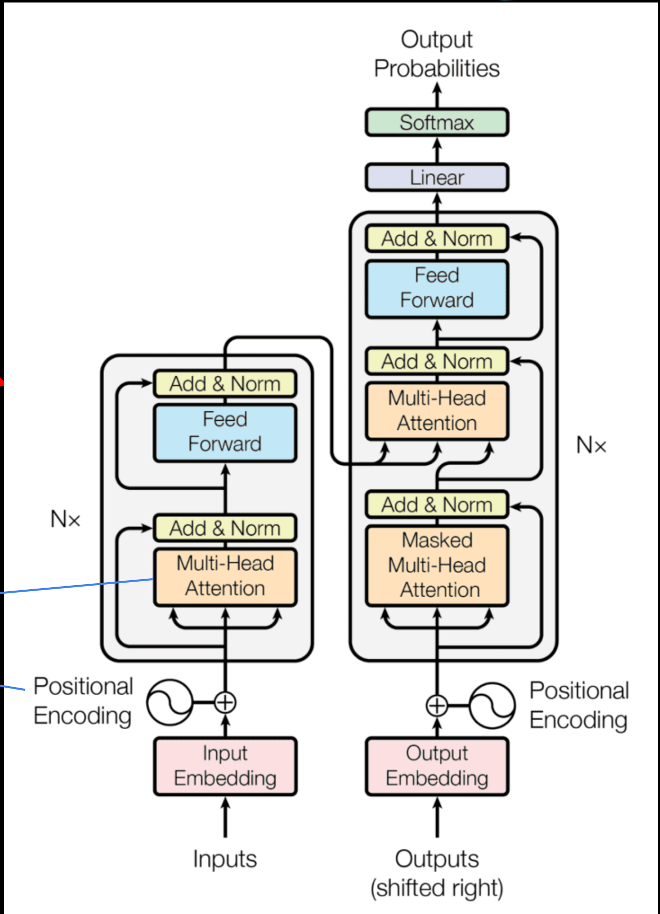
Diffusion Policy

Transformers



The core model is usually a transformer-based architecture (ACT, OCTO, OpenVLA, …)

1. Positional Encoding
2. Self-Attention



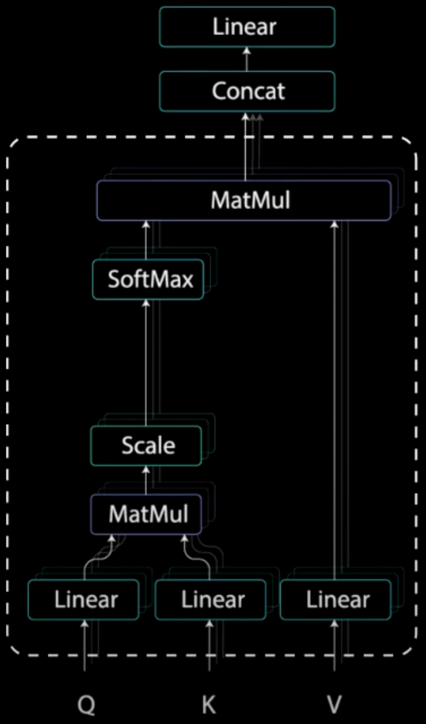
Positional Embeddings

- Unlike RNNs, transformers don't inherently understand sequence order Positional embeddings add information about token positions in sequences.

- Positional information is added to token embeddings, often using sinusoidal functions to allow model generalization to various sequence lengths.

$$PE_{pos,2i} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \quad PE_{pos,2i+1} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

Attention

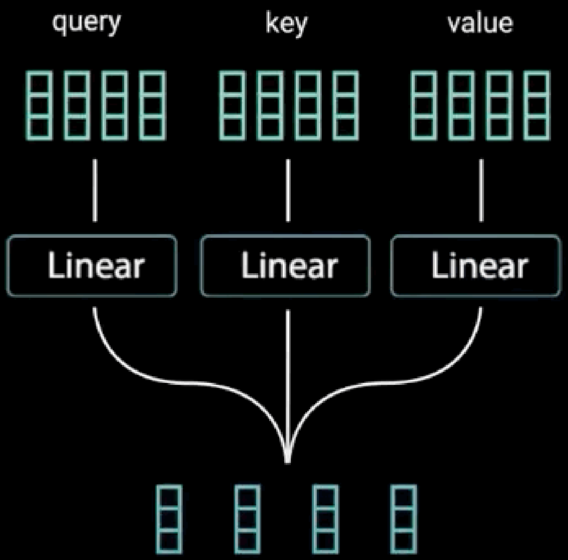


From input embeddinsg use 3 distinct linear layers to create query, key and value vectors

"For example, when you type a query to search for some video on Youtube, the search engine will map your **query** against a set of **keys** (video title, description etc.) associated with candidate videos in the database, then present you the best matched videos (**values**)."

ETH zürich SoftRobotics Laboratory

Attention

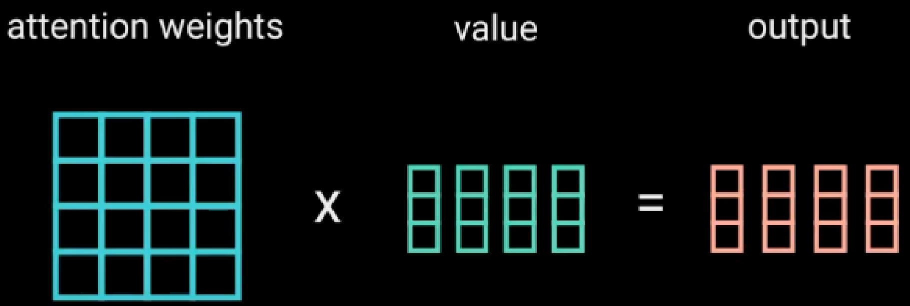


	Hi	how	are	you
Hi	98	27	10	12
how	27	89	31	67
are	10	31	91	54
you	12	67	54	92

Attention

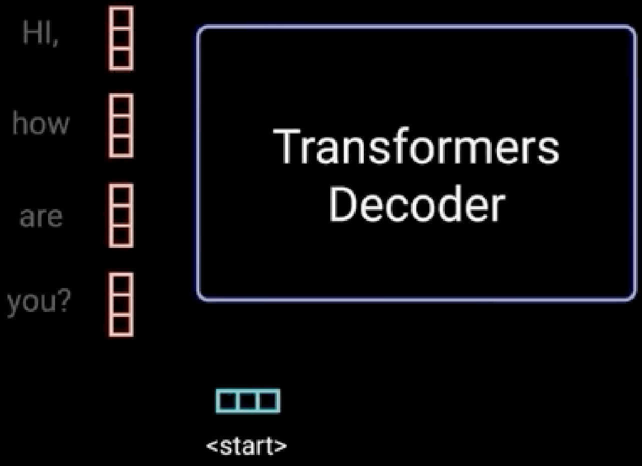
Softmax( ) =

	Hi	how	are	you
Hi	0.7	0.1	0.1	0.1
how	0.1	0.6	0.2	0.1
are	0.1	0.3	0.6	0.1
you	0.1	0.3	0.3	0.3

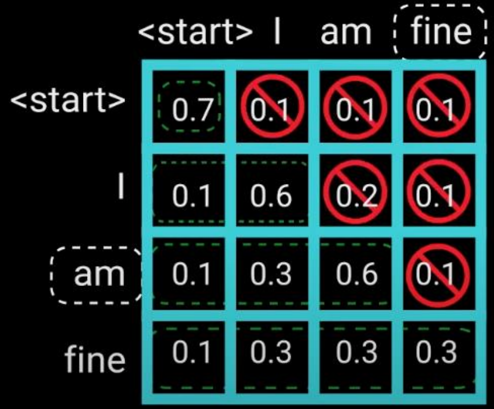


$$softmax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j))}$$

Decoder



The decoder is autoregressive
-> How to prevent conditioning on future tokens?



Scaled Scores + Look-Ahead Mask = Masked Scores

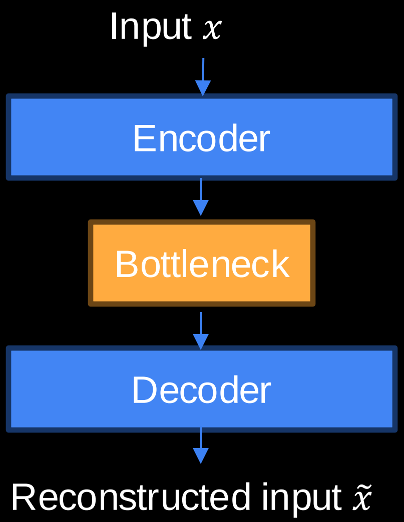
$$\begin{bmatrix} 0.7 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.6 & 0.2 & 0.1 \\ 0.1 & 0.3 & 0.6 & 0.1 \\ 0.1 & 0.3 & 0.3 & 0.3 \end{bmatrix} + \begin{bmatrix} 0 & -inf & -inf & -inf \\ 0 & 0 & -inf & -inf \\ 0 & 0 & 0 & -inf \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.7 & -inf & -inf & -inf \\ 0.1 & 0.6 & -inf & -inf \\ 0.1 & 0.3 & 0.6 & -inf \\ 0.1 & 0.3 & 0.3 & 0.3 \end{bmatrix}$$

Transformer explainer

- https://poloclub.github.io/transformer-explainer/

Variational Autoencoder

- Autoencoder

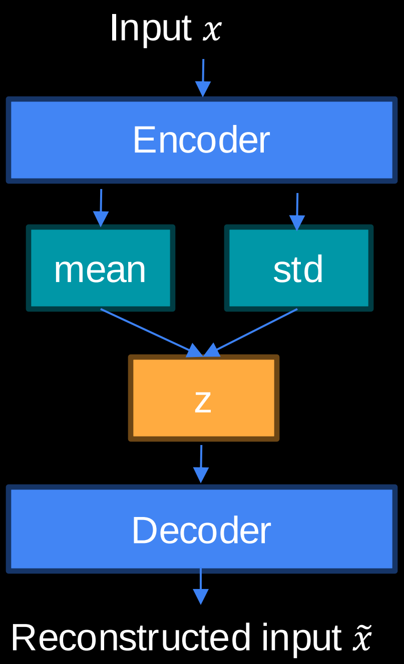



```
def forward(self, x):
    z = self.encode(x)
    return self.decode(z)

def loss_function(recon_x, x):
    l2 = F.mse_loss(recon_x, x, reduction='sum')
    return l2
```

ETH zürich SoftRobotics Laboratory

Variational Autoencoder

- Variational Autoencoder



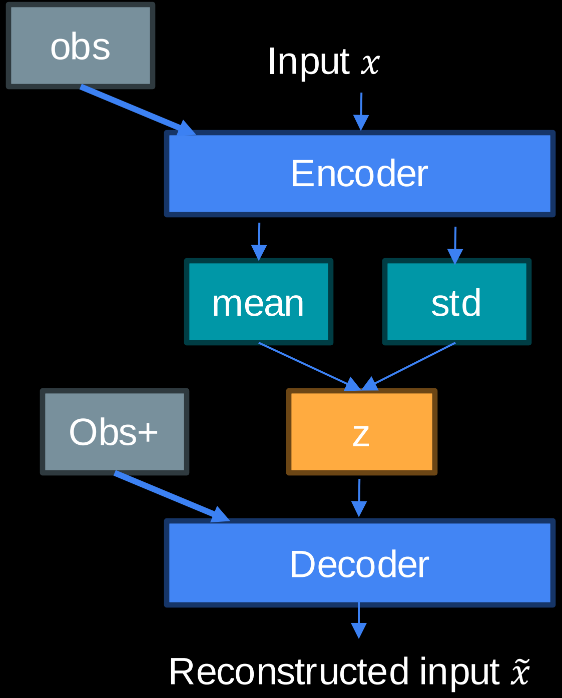

```
def reparameterize(self, mu, logvar):
    std = torch.exp(0.5 * logvar)
    eps = torch.randn_like(std)
    return mu + eps * std

def forward(self, x):
    mu, logvar = self.encode(x)
    z = self.reparameterize(mu, logvar)
   return self.decode(z), mu, logvar

def loss_function(recon_x, x, mu, logvar):
    kld = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    BCE = F.mse_loss(recon_x, x, reduction='sum’)
    return BCE
```

ETH zürich SoftRobotics Laboratory

Link: [INTERPRETING LATENT SPACE AND BIAS]()

Conditional Variational Autoencoder

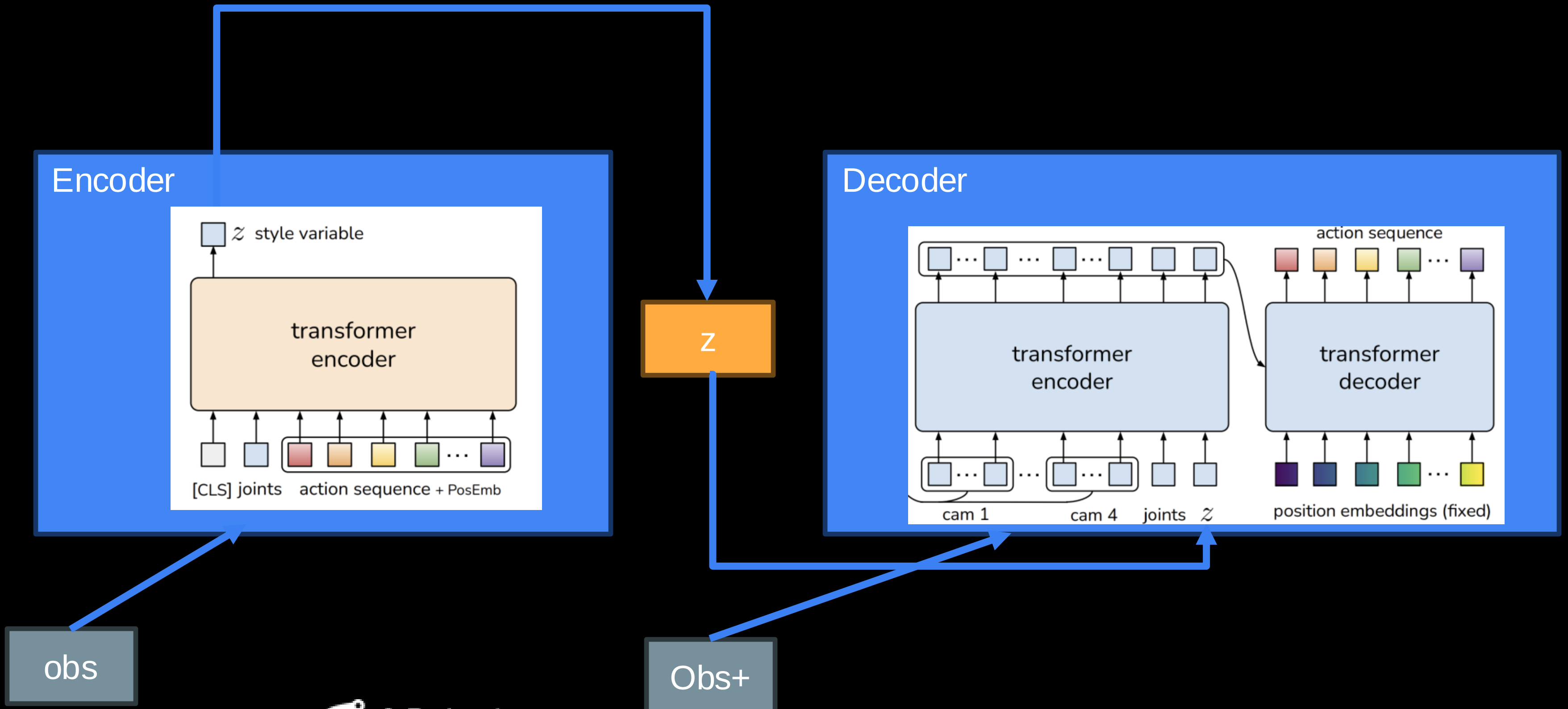
- Variational Autoencoder




```
def reparameterize(self, mu, logvar):
    std = torch.exp(0.5 * logvar)
    eps = torch.randn_like(std)
    return mu + eps * std

def forward(self, x, obs):
    mu, logvar = self.encode(x)
    z = self.reparameterize(mu, logvar)
    return self.decode(z, self.obs_encode(obs))

def loss_function(recon_x, x, mu, logvar):
    kld = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    BCE = F.mse_loss(recon_x, x, reduction='sum’)
    return BCE
```

ETH zürich SoftRobotics Laboratory

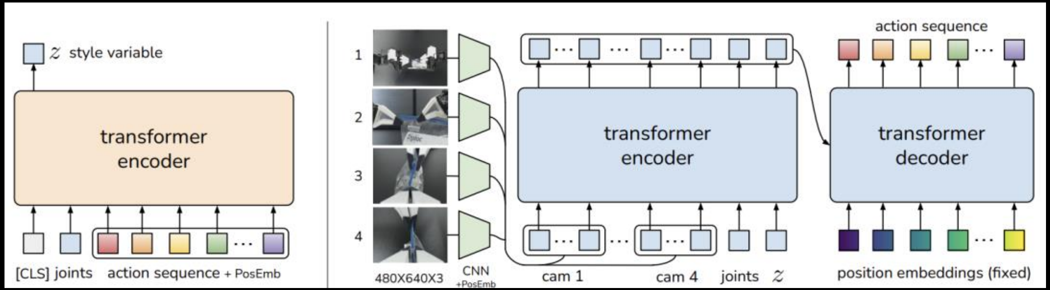
Action Chunking Transformers



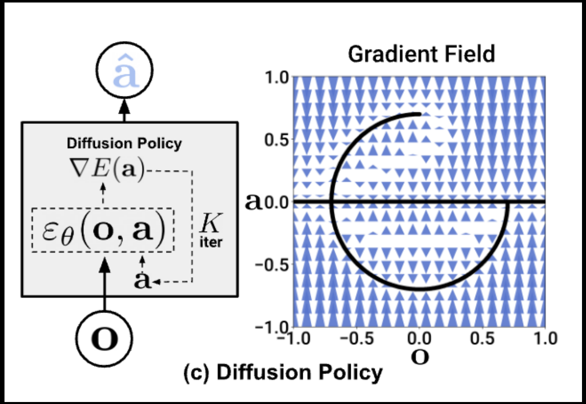
Contents

What we need to understand today:

1. Transformer Architecture
2. Variational Autoencoder
3. Diffusion Mechanism



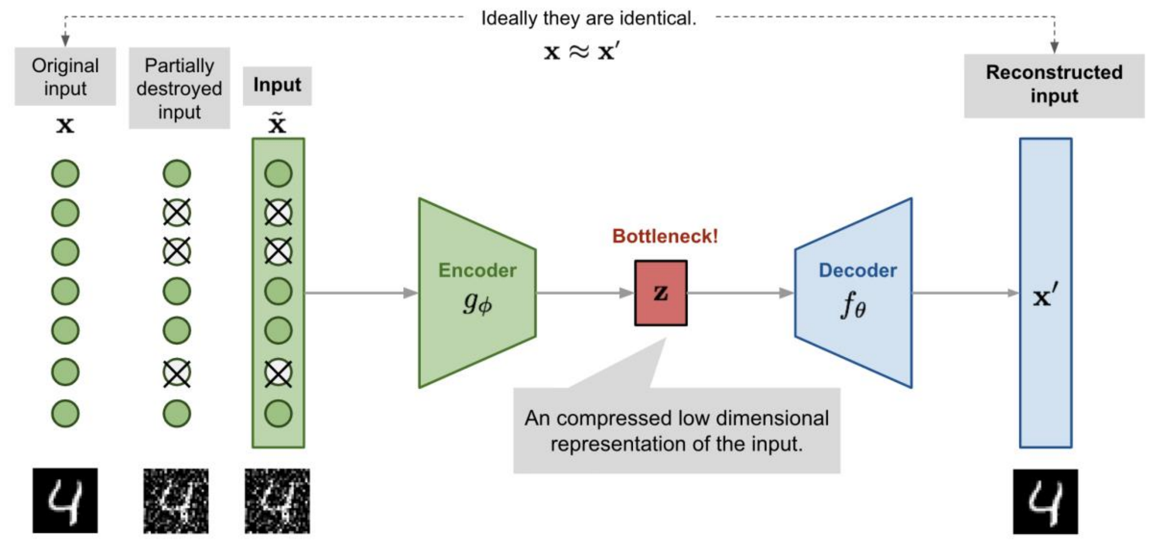
ACT



Diffusion Policy

Denoising

Besides compressing, vae models can be trained to recover the original input.

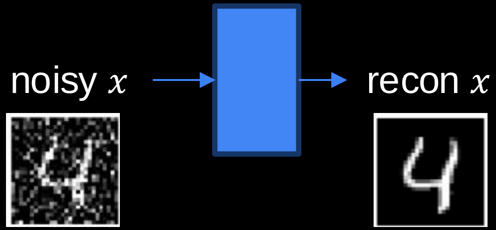



```
def train_iter(self, x):
    x_noise = x + torch.rand_like(x)
    recon_x = self.forward(x)
    loss = F.mse_loss(x, recon_x)
    loss.backward()
```

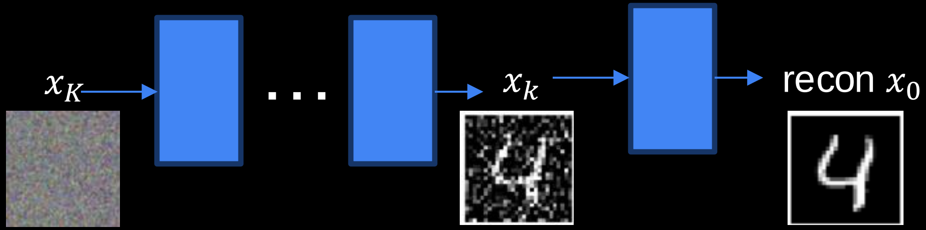
Resource: https://lilianweng.github.io/posts/2018-08-12-vae/

Diffusion Models

- Given a denoise model



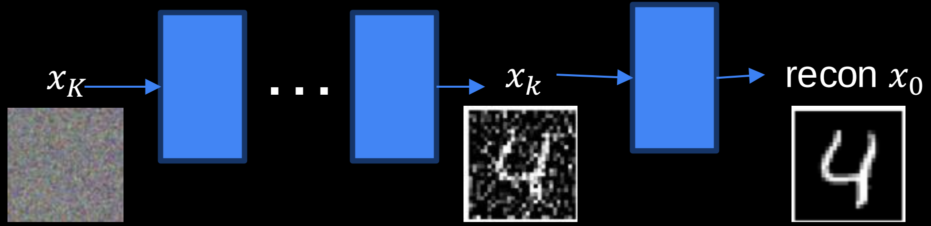
- What about stacking them many times



ETH zürich SoftRobotics Laboratory

Diffusion Models

- We stack many denoise steps to reconstruct from gaussian noise



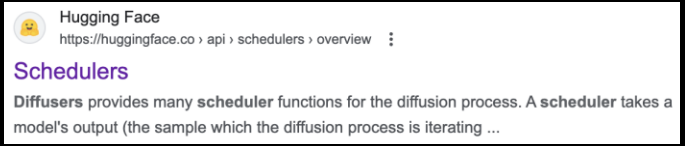
- How to train the network
  - aka: how to define the loss
    - aka: how to find x_k and x_{k+1}

Loss=MSE(x_k $\widetilde{x_k}$)

- x_{k+1} is can be computed by adding noise from x_k (**Forward diffusion process**)

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

- But we only have x_0
  - Good news: x_{k+1}can be directly computed from x_0



ETH zürich SoftRobotics Laboratory

Diffusion Models: Train with Scheduler

```
noise_scheduler = DDPMScheduler(num_train_timesteps=1000)
def train_iter(batch):
    clean_images = batch["images"]
    # Sample noise to add to the images
    noise = torch.randn(clean_images.shape).to(clean_images.device)
    bs = clean_images.shape[0]
    # Sample a random timestep for each image
    timesteps = torch.randint(
        0, noise_scheduler.config.num_train_timesteps, (bs,),
    ).long()
    # Add noise to the clean images according to the noise magnitude at each timestep
    # (this is the forward diffusion process)
    noisy_images = noise_scheduler.add_noise(clean_images, noise, timesteps)
    # Predict the noise residual
    noise_pred = model(noisy_images, timesteps)
    loss = F.mse_loss(noise_pred, noise)
```

Diffusion Models: Generate with Scheduler

```
# Initialize random noise as the starting point for generation
target = torch.randn((num_samples,) + image_shape)
# Iteratively remove noise step by step
for t in noise_scheduler.timesteps:
   # Model prediction for noise residuals
   with torch.no_grad():
      noise_pred = model(target, t).long())
   # Compute the previous image state by reversing the diffusion process
   target = noise_scheduler.step(noise_pred, t, target).prev_sample
```

srl_il