COMPARING ARCHITECTURES FOR REACHABILITY

Contents

1. Shared architecture blocks

1.1. **ResidualBlock.** "Residual" because in a residual block, the network only has to learn the difference (the "residual") between the input and the output: $y = x + f(x)$. If the optimal transformation is just the identity, the weights can just shrink toward zero, and x will pass through untouched.

1.1.1. *Vanishing gradient problem.* Residual blocks solve the vanishing gradient problem. In a standard deep network without skip connections, each layer is a function $f(x)$. A deep network looks like:

$$y = f_\ell(f_{\ell-1}(\cdots f_1(x)))$$

Backprop-derived weight update for the first layer f_1:

$$\frac{\partial \mathcal{L}}{\partial f_1} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial f_\ell}{\partial f_{\ell-1}} \cdots \frac{\partial f_2}{\partial f_1}$$

If the weights are small, these derivatives are often < 1. Multiplying many numbers < 1 causes the gradient to shrink exponentially, s.t. by the time it reaches the early layers (e.g., f_1), the gradient is essentially zero, meaning those layers never learn.

Residual blocks fix this problem by changing the layer math from $y = f(x)$ to $y = x + f(x)$. Backpropagation for one layer then becomes:

$$\frac{d}{dx}[x + f(x)] = 1 + \frac{df}{dx}$$

Date: January 26, 2026.

Because of the +1, the gradient of the loss can flow through the identity path (the skip connection) completely undiminished, regardless of weights in the "weight path."

1.1.2. *Number of parameters.*

(1) LayerNorm(d_model): 2 parameters
- LayerNorm computes the mean μ and variance σ^2 across all features of the current sample, and then standardizes this sample's features to have zero mean and unit variance:
$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$
LayerNorm introduces $\underline{2 \cdot \text{d\_model}}$ learnable parameters, γ and β, to allow the network to "undo" the normalization if it finds that the original distribution was actually better for learning:
$$y_i = \gamma \hat{x}_i + \beta$$

(2) Linear(d_model, d_model): d_model $\times$ d_model + d_model parameters

(3) SiLU: 0 parameters (activation function)
$$\text{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}$$

(4) Dropout: 0 parameters

(5) Linear(d_model, d_model): d_model $\times$ d_model + d_model parameters

Total: $\boxed{P_{\text{ResBlock}}(d) = 2d \cdot (d + 2)}$ parameters

1.2. **ResidualMLP.**

1.2.1. *Number of parameters.*

(1) Input projection: $W_1 : \mathbb{R}^{\text{in\_dim} \times \text{hidden\_dim}} \Rightarrow$ in_dim $\times$ hidden_dim + hidden_dim parameters

(2) Residual blocks: $W_2 \ldots W_{\text{num\_blocks}+1}$: num_blocks $\times$ $P_{\text{ResBlock}}(d = \text{hidden\_dim})$ parameters

(3) Output projection: LayerNorm(hidden_dim) + SiLU + Linear(hidden_dim, out_dim) = 2 $\times$ hidden_dim + hidden_dim $\times$ out_dim + out_dim parameters

Denoting in_dim $= d_{\text{in}}$, out_dim $= d_{\text{out}}$, z_dim $= d_z$, hidden_dim $= d_h$, and num_blocks $= n_b$:

$$\text{Total parameters: } \boxed{P_{\text{ResMLP}}(d_{\text{in}}, d_h, d_{\text{out}}, d_z, n_b) = d_h(d_{\text{in}} + 2n_b(d_h + 2) + d_{\text{out}} + 3) + d_{\text{out}}}$$

2. Conditional VAE

2.1. **Architecture.**

- Encoder: ResidualMLP(in_dim = d_c_feat + d_q_feat, hidden_dim = hidden_dim, out_dim = 2 $\cdot$ z_dim, num_blocks = num_blocks)
$$E : \begin{bmatrix} C_{\text{feat}} \\ Q_{\text{feat}} \end{bmatrix} \to \begin{bmatrix} \hat{\mu}(z) \\ \widehat{\text{logvar}}(z) \end{bmatrix}$$
Given feature vector Q_{feat} and conditioning vector C_{feat} for some data point x, the encoder produces a normal distribution $q(z \mid C_{\text{feat}}, Q_{\text{feat}})$.
- Decoder: ResidualMLP(in_dim = d_c_feat + z_dim, hidden_dim = hidden_dim, out_dim = 2 $\cdot$ d_q_feat, num_blocks = num_blocks)
$$D : \begin{bmatrix} C_{\text{feat}} \\ z \end{bmatrix} \to \begin{bmatrix} \hat{\mu}(Q_{\text{feat}}) \\ \widehat{\text{logvar}}(Q_{\text{feat}}) \end{bmatrix}$$
Given latent vector z and conditioning vector C_{feat}, the decoder produces a normal distribution $q(Q_{\text{feat}} \mid C_{\text{feat}}, z)$ (plausible Q_{feat}'s that would correspond to latent z given C_{feat}). Would it be possible for the decoder to output a mixture of Gaussians instead?
- VAE:
$$\text{Data point } (q, c, h) \xrightarrow{\text{featurize}} q_{\text{feat}}, c_{\text{feat}} \xrightarrow{E} \hat{\mu}(z), \widehat{\text{logvar}}(z) \Rightarrow z \sim \mathcal{N}(\hat{\mu}(z), \widehat{\text{logvar}}(z))$$
$$z, c_{\text{feat}} \xrightarrow{D} \hat{\mu}(q_{\text{feat}}), \widehat{\text{logvar}}(q_{\text{feat}}) \Rightarrow \boxed{\hat{q}_{\text{feat}}} \sim \mathcal{N}(\hat{\mu}(q_{\text{feat}}), \widehat{\text{logvar}}(q_{\text{feat}}))$$

2.1.1. *Sampling.* To sample from the cVAE, we only use the decoder portion:

$$h_{\text{world}}, c_{\text{feat}}, z \sim \mathcal{N}(0,1) \xrightarrow{D} \hat{\mu}(q_{\text{feat}}), \widehat{\text{logvar}}(q_{\text{feat}}) \Rightarrow \hat{q}_{\text{feat}} \sim \mathcal{N}(\hat{\mu}(q_{\text{feat}}), \widehat{\text{logvar}}(q_{\text{feat}}))$$

2.1.2. *Number of parameters.*

- Encoder: $P_{\text{ResMLP}}(d_{\text{in}} = d_{cf} + d_{qf}, d_h = d_h, d_{\text{out}} = 2d_z, n_b = n_b)$ where d_{cf} = d_c_feat and d_{qf} = d_q_feat.
$$P_{\text{enc}} = d_h((d_{cf} + d_{qf}) + 2n_b(d_h + 2) + 2z + 3) + 2z$$
- Decoder: $P_{\text{ResMLP}}(d_{\text{in}} = d_{cf} + d_z, d_h = d_h, d_{\text{out}} = 2d_{qf}, n_b = n_b)$
$$P_{\text{dec}} = \boxed{d_h((d_{cf} + d_z) + 2n_b(d_h + 2) + 2d_{qf} + 3) + 2d_{qf}}$$

Only the decoder participates in generation, so we have only P_{dec} parameters at inference.

2.2. **Loss function.**

- Reconstruction loss $\mathcal{L}_{\text{NLL}}$:
$$\mathcal{L}_{\text{NLL}} = -\log[p(q_{\text{feat}} \mid \hat{\mu}(q_{\text{feat}}), \hat{\sigma}(q_{\text{feat}})^2)]$$
where q_{feat} is obtained from featurizing a true data point q, and $\hat{\sigma}(q_{\text{feat}})^2)$ are outputted by the VAE. Aka, we want the model to predict a Gaussian that tightly centers around the true q_{feat}
- KL loss $\mathcal{L}_{\text{KL}}$:
$$\mathcal{L}_{\text{KL}} = D_{\text{KL}}(\mathcal{N}(\hat{\mu}(z), \hat{\sigma}(z)^2) \,||\, \mathcal{N}(0,1))$$
Aka., we want the latent distribution over z to resemble that of a standard Gaussian.
- Forward kinematics MSE loss:
$$||\text{FK}(\hat{\mu}(q_{\text{feat}})) - h||_2^2$$
where h is the ground truth end effector target paired with q (that was fed through VAE to obtain model estimate $\hat{\mu}(q_{\text{feat}})$).

Combined:

$$\mathcal{L} = \mathcal{L}_{\text{NLL}} + \beta \cdot \mathcal{L}_{\text{KL}} + \lambda_k \cdot \mathcal{L}_{\text{fk\_mse}}$$

2.3. **Strengths vs. weaknesses.**

2.3.1. *Strengths.*

2.3.2. *Weaknesses.*

2.4. **Empirical performance.**

2.4.1. *2D 2-link rotary arm robot.*

3. Conditional INN (normalizing flow)

3.1. **Architecture.**

3.1.1. *Subparts.*

- Subnet: ResidualMLP(in_dim, hidden_dim, out_dim, num_blocks)
$$S : \begin{bmatrix} C_{\text{feat}} \\ \frac{1}{2}Q_{\text{feat}} \end{bmatrix} \to \begin{bmatrix} s \\ t \end{bmatrix}$$
Given half of feature vector Q_{feat} (coupling block splits the input into two halves) and conditioning vector C_{feat}, the subnet predicts affine transformation parameters (scaling and translation). The subnet is the internal neural network used within the coupling blocks.
- Conditioning node provides the context C_{feat} to every coupling block in the network.
- Coupling blocks: ($\times$ num_blocks)
  1. Permuting node: Shuffles the dimensions to ensure all features interact.
  2. ActNorm: $y = s \odot x + b$ where s is the scale vector and b is the bias vector. Both have the same dimensionality as the number of features = d_q_feat.
     - LayerNorm: Across the features, for each sample, calculate μ and σ^2 using all features of that sample.

– ActNorm: Initialize s and b by computing the mean and variance of the first batch. For every subsequent batch, s and b are treated as regular trainable parameters, so $y_i = s_i \cdot x_i + b_i \Rightarrow y_i$ does not look at x_{i+1} or x_j, so the Jacobian matrix of ActNorm is diagonal.

Normalizing flows use ActNorm instead of LayerNorm because LayerNorm makes the Jacobian dense: in LayerNorm, every feature in the output depends on every feature in the input.

(3) Affine coupling:

– **Vanilla affine coupling:** Splits the input into two halves $\left(Q^{(1)}_{\text{feat}}, Q^{(2)}_{\text{feat}}\right) := (q_1, q_2)$. One half remains unchanged, while the other is transformed:

$$q_1' = q_1$$
$$q_2' = q_2 \odot \exp(s(q_1, C_{\text{feat}})) + t(q_1, C_{\text{feat}})$$

– `Fm.GLOWCouplingBlock` **affine coupling:** Important note: GLOWCouplingBlock spawns *two distinct sub-networks* to predict two sets of affine transformation parameters every coupling block layer. It splits the input into (q_1, q_2) and applies a two-step alternating update:

$$q_2' = q_2 \odot \exp(s_1(q_1, C_{\text{feat}})) + t_1(q_1, C_{\text{feat}}$$
$$q_1' = q_1 \odot \exp(s_2(s_2', C_{\text{feat}})) + t_2(q_2', C_{\text{feat}}$$

We use GLOWCouplingBlock in our implementation.

3.1.2. *Compiled architecture.* Essentially, each layer with a coupling block applies a learned affine transformation to some permutation of the full input feature vector. The permutations are deterministic across training and inference: every time the model sees a sample, the “shuffle” for block k is identical, allowing the subnets to learn which specific dimensions of the input vector cotnain the information needed to predict the scaling and translation for other dimensions. The full transformation is the combined effect of all these affine transformations: all layers f_i compile into invertible function f:

$$f = f_k \circ \ldots \circ f_2 \circ f_1$$

Every f_i (total num_blocks number of f_i) consists of:

- Permutations
- ActNorm
- Coupling block (ResidualMLP)

↑ Forward mapping:

$$f : (q_{\text{feat}}, c_{\text{feat}}) \to (z, \log|\det J|)$$

- $z \sim \mathcal{N}(0, I)$ is encouraged during training.

Reverse mapping (generation/sampling):

$$f^{-1} : (z, c_{\text{feat}}) \to q_{\text{feat}}$$

- f^-1 is definitely derivable from f because the network for f was designed to be invertible.

To sample a robot configuration q_{feat} given conditional vector c_{feat}:

$$z \sim \mathcal{N}(0, I) \Rightarrow \boxed{\hat{q}_{\text{feat}}} = f^{-1}(z, c_{\text{feat}})$$

3.1.3. *Number of parameters.* Each invertible block involves:

- ActNorm(d_q_feat): $2d_{qf}$ parameters
  - `nodes.append(Ff.Node(nodes[-1], Fm.ActNorm, {}))` applies ActNorm to the data path only (not the conditioning node).
- GLOWCouplingBlock(d_c_feat + d_q_feat): 2 × ResidualMLP(in_dim = d_c_feat + d_q_feat/2, hidden_dim = hidden_dim, out_dim = 2 · (d_q_feat/2), num_blocks = num_subnet_blocks [:= n_{snb}])
  - Subnet outputs s and $b \Rightarrow 2 \cdot$ (d_q_feat/2) = d_q_feat

$$\begin{aligned} P_{\text{subnet}} &= P_{\text{ResMLP}}(d_{\text{in}} = d_{cf} + 0.5 d_{qf}, d_h = d_h, d_{\text{out}} = d_{qf}, n_b = n_{snb}) \\ &= d_h(d_{cf} + 1.5 d_{qf} + 2 n_{snb}(d_h + 2) + 3) + d_{qf} \\ P_{\text{GLOW}} &= 2 \cdot P_{\text{subnet}} \end{aligned}$$

Total parameters:

$$P_{\text{cINN}} = \underbrace{n_b}_{\text{Blocks}} \cdot \left[\underbrace{2d_{qf}}_{\text{ActNorm}} + \underbrace{2}_{\substack{\text{GLOW}\\\text{Multiplier}}} \cdot \left(\underbrace{d_h(d_{cf} + 0.5d_{qf} + 2n_{snb}(d_h + 3) + 3) + d_{qf}(d_h + 2)}_{\text{Subnet (ResidualMLP)}} \right) \right]$$

3.2. **Loss function.**

- NLL loss: We want to optimize the model to predict maximal probability for true data points, i.e. maximize $p_X(x)$. If $z = f_\theta(x)$ and f_θ is bijective (invertible) and differentiable, densities transform via:

$$p_X(x) = p_Z(f_\theta(x)) \cdot \left| \det \frac{\partial f_\theta}{\partial x} \right|$$

  Intuition behind this equation for $p_X(x)$: to maximize $p_X(x)$, we can either increase $p_Z(f_\theta(x))$ (map x to the center of the Gaussian) and/or make $|\det \frac{\partial z}{\partial x}|$ large, i.e. make it s.t. a small change in x leads to a large change in z (expand z coordinates s.t. x covers a larger portion of the Gaussian's probability mass). Taking negative log:

$$-\log p_X(x) = -\log p_Z(z) - \log|\det J_f(x)| \qquad z = f_\theta(x), J_f(x) = \frac{\partial f_\theta(x)}{\partial x}$$

  If $p_Z = \mathcal{N}(0, I)$, then $-\log p_Z(z) = \frac{1}{2}||z||^2 + \text{const}$. The goal is to model an invertible function from $p_Z \mapsto p_X$ s.t. $p_X(x) = p_Z(f_\theta(x))$ is big for real data x and small/near-zero for data points outside of the true data distribution. Thus, the training objective per sample (real data point x) becomes:

$$\mathcal{L}_{\text{NLL}} = \tfrac{1}{2}||z||^2 - \log|\det J_f(x)| + \text{const}$$

  Minimizing this loss is equivalent to maximizing $p_X(x)$.
  - *What prevents the model from assigning high probability to the entire space?* We want to maximize $p_X(x)$, but we also want to minimize $p_X(y \notin X)$. The NLL objective appears to only enforce maximizing $p_X(x)$!
    - The "pressure" on the model to avoid assigning high probability everywhere comes from the incompressibility of probability: by definition, a PDF must integrate to 1:

$$\int p_X(x)\,dx = 1$$

      Because the total area under the curve is fixed at 1, probability is a finite resource: if the model increases the probability $p_X(x)$ for a true sample, it *must* decrease the probability somewhere else.
    - $\uparrow$ The Jacobian term enforces this incompressibility of probability in the loss function: $|\det J|$ measures how much the function f_θ stretches or compresses space at point x.
      - To increase probability: The model needs to "squeeze" the space around a true data point x so that a blob in the X (bubble of feasible configs around x) space maps to a very small, high-density region in the Z (latent) space (near the origin, where the Gaussian p_Z is highest).
      - The model cannot "squeeze" the entire input space to the origin to give everything high probability, because space is finite. The model is a bijection, so it has to map the "empty" spaces in the data distribution to the "empty" (low-density) tails of the Gaussian distributionn. Expanding one region of probability space entails compressing another region. This prevents the model from predicting high probability for everything!

- Forward kinematics MSE: Using the current state of the model f to predict $\hat{q} = f^{-1}(z, c)$ (where $z \sim \mathcal{N}(0, I)$), we can compute
$$||\mathrm{FK}(\hat{q}) - h||_2^2$$
for data points (q, c, h).

Combined loss:
$$\mathcal{L} = \mathcal{L}_{\mathrm{NLL}} + \lambda_k \mathcal{L}_{\mathrm{fk\_mse}}$$

3.3. **Strengths vs. weaknesses.**

3.4. **Empirical performance.**

4. Conditional diffusion

4.1. **Architecture.** The model architecture, ResMLPDenoiser, is used to model $\epsilon_\theta(x, t)$ (predict noise ϵ_t given x_t, t, allowing us to define $p_\theta(x_{t-1} \mid x_t)$ and denoise by one step).

ResMLPDenoiser consists of a MLP (2 linear layers with Mish activation) for processing sinusoidal time embeddings and a ResidualMLP that inputs the processed time embeddings + q_{feat} + c_{feat} to output the predicted noise ϵ. Here, in usual diffusion moidel notation, $x = (q_{\mathrm{feat}}, c_{\mathrm{feat}})$.

4.1.1. *Number of parameters.*

(1) Time embedding: time_dim = d_t = hidden_dim // 4
  - Linear layer: $d_t \times 2d_t + 2d_t$
  - Mish layer: Activation $f(x) = x \cdot \tanh(\ln(1 + e^x))$
    - Mish is slower than SiLU (Swish), but is "smoother" in its top-order derivatives (which is cited as the reason it helps with training stability in very deep architectures)
  - Linear layer: $2d_t \times d_t + d_t$

(2) ResidualMLP(in_dim = $d_q + d_c + d_t$, hidden_dim = d_h, out_dim = d_q, num_blocks = n_b, dropout = dropout)

Total parameters:
$$P_{\mathrm{cDiff}} = \boxed{\underbrace{4d_t^2 + 3d_t}_{\text{Time MLP}} + \underbrace{d_h(2d_{qf} + d_{cf} + d_t + 2n_b(d_h + 2) + 3) + d_{qf}}_{\text{ResidualMLP}}}$$

4.2. **Loss function.** Using lots of algebra, we can show that ELBO loss for maximizing the log-likelihood of the data $\log p_\theta(x_0)$ ends up being proportional to noise MSE:
$$\mathcal{L} \propto ||\epsilon - \epsilon_\theta(x_t, t)||^2$$
We train on this MSE loss.

4.3. **Strengths vs. weaknesses.**

4.4. **Empirical performance.**

5. Comparing architectures

5.1. **Number of parameters.** Total number of parameters in the model:

Model	Total parameter count
cVAE	$d_h\big((d_{cf} + d_{qf}) + 2n_b(d_h + 2) + 2d_z + 3\big) + 2d_z + d_h\big((d_{cf} + d_z) + 2n_b(d_h + 2) + 2d_{qf} + 3\big) + 2d_{qf}$
cINN	$n_b \cdot [2d_{qf} + 2 \cdot (d_h(d_{cf} + 0.5d_{qf} + 2n_{snb}(d_h + 3) + 3) + d_{qf}(d_h + 2))]$
cDiffusion	$(4d_t^2 + 3d_t) + d_h\big(2d_{qf} + d_{cf} + d_t + 2n_b(d_h + 2) + 3\big) + d_{qf}$

Counting only parameters available at inference:

6. Problem Description

Our goal is to learn *where the robot should stand* given a desired grasp.

Model	Inference-only parameter count
cVAE	$d_h\big((d_{cf}+d_z)+2n_b(d_h+2)+2d_{qf}+3\big)+2d_{qf}$
cINN	$n_b \cdot \big[3d_{qf}+d_h\big(d_{cf}+1.5d_{qf}+2n_{snb}(d_h+2)+3\big)\big]$
cDiffusion	$(4d_t^2+3d_t)+d_h\big(2d_{qf}+d_{cf}+d_t+2n_b(d_h+2)+3\big)+d_{qf}$

6.1. **Problem inputs.** Formulating the problem, we are given as inputs:

- $x_e \in SE(3)$: desired end-effector pose in world coordinates
- $b \in SE(2)$: wheeled mobile base pose (x, y, γ)
- $q \in \mathbb{R}^n$: arm joint configuration.

6.2. **Problem outputs.** We want to find a range of feasible whole-body configurations that satisfy the following requirements:

(1) **IK feasibility:** forward kinematics$(b, q) = x_e$
(2) **Visibility/sensing constraints:** The end-effector should lie within the camera's field of view and range (and un-occluded by the robot itself), i.e. visible$(b, q, x_e) = 1$
(3) **Ball of feasible configurations:** We want a region in joint space around (b, q) that keeps the hand in approximately the right pose and satisfies the above constraints.

Our target is to model some distribution $p(b, q \mid x_e)$ subject to these constraints $\uparrow$, ideally with some notion of pose quality (visibility, manipulability, clearance, etc.).