Aalto University

CS-E4890: Deep Learning
Flow-based generative models

Alexander Ilin

Invertible generative process

- In flow-based generative models, the generative process is usually defined as

$$\mathbf{z} \sim p_{\theta}(\mathbf{z})$$
$$\mathbf{x} = \mathbf{g}_{\theta}(\mathbf{z})$$

  where $\mathbf{z}$ is the latent variable and $p_{\theta}(\mathbf{z})$ has a simple tractable density, such as a spherical multivariate Gaussian distribution: $p_{\theta}(\mathbf{z}) = \mathcal{N}(\mathbf{z}; 0, \mathbf{I})$.
- The function $\mathbf{g}_{\theta}$ is invertible (*bijective*).
- Inference is done by $\mathbf{z} = \mathbf{f}_{\theta}(\mathbf{x}) = \mathbf{g}_{\theta}^{-1}(\mathbf{x})$.

Invertible generative process

- In flow-based generative models, the generative process is usually defined as

$$\mathbf{z} \sim p_{\boldsymbol{\theta}}(\mathbf{z})$$
$$\mathbf{x} = \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{z})$$

  where $\mathbf{z}$ is the latent variable and $p_{\boldsymbol{\theta}}(\mathbf{z})$ has a simple tractable density, such as a spherical multivariate Gaussian distribution: $p_{\boldsymbol{\theta}}(\mathbf{z}) = \mathcal{N}(\mathbf{z}; 0, \mathbf{I})$.
- The function $\mathbf{g}_{\boldsymbol{\theta}}$ is invertible (*bijective*).
- Inference is done by $\mathbf{z} = \mathbf{f}_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{g}_{\boldsymbol{\theta}}^{-1}(\mathbf{x})$.

- Compare to the generative model that we considered with variational autoencoders:

$$\mathbf{z} \sim p_{\boldsymbol{\theta}}(\mathbf{z})$$
$$\mathbf{x} = \mathbf{g}_{\boldsymbol{\theta}}(\mathbf{z}) + \varepsilon$$

  - $\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{z})$ was not generally invertible
  - one could add extra noise ε.
- Because of this, it was not possible to recover $\mathbf{z}$ from $\mathbf{x}$ easily. We had to design an inference procedure that involved approximations $q(\mathbf{z}) \approx p(\mathbf{z} \mid \mathbf{x}, \boldsymbol{\theta})$.

Normalizing flows

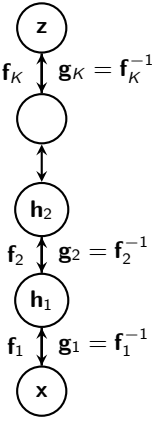
- Flow-based generative models use invertible $\mathbf{g}_\theta$:

$$\mathbf{z} \sim p_\theta(\mathbf{z})$$
$$\mathbf{x} = \mathbf{g}_\theta(\mathbf{z})$$

- To implement this idea, we need to construct an invertible transformation $\mathbf{x} = \mathbf{g}_\theta(\mathbf{z})$, $\mathbf{z} = \mathbf{f}_\theta(\mathbf{x}) = \mathbf{g}_\theta^{-1}(\mathbf{x})$.
- We can do so by constructing a sequence of invertible transformations:

$$\mathbf{x} \underset{\mathbf{g}_1}{\overset{\mathbf{f}_1}{\longleftrightarrow}} \mathbf{h}_1 \underset{\mathbf{g}_2}{\overset{\mathbf{f}_2}{\longleftrightarrow}} \mathbf{h}_2 \cdots \underset{\mathbf{g}_K}{\overset{\mathbf{f}_K}{\longleftrightarrow}} \mathbf{z}$$

- Such a sequence of invertible transformations is called a (normalizing) flow (Rezende and Mohamed, 2015).



Learning flow-based generative models

- We can tune the parameters of the model by maximizing the log-likelihood

$$\mathcal{F}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \log p_{\boldsymbol{\theta}}(\mathbf{x}_i)$$

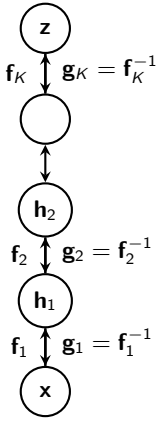
- Since mapping $\mathbf{x} \to \mathbf{z}$ is invertible, we can use the change-of-variables rule:

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = p_{\boldsymbol{\theta}}(\mathbf{z}) \left| \det \frac{\partial \mathbf{z}}{\partial \mathbf{x}} \right|$$

- This yields the log-likelihood for a single datapoint $\mathbf{x}$:

$$\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log p_{\boldsymbol{\theta}}(\mathbf{z}) + \log \left| \det \frac{\partial \mathbf{z}}{\partial \mathbf{x}} \right| = \log p_{\boldsymbol{\theta}}(\mathbf{z}) + \sum_{k=1}^{K} \log \left| \det \frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}} \right|$$

where $d\mathbf{h}_k / d\mathbf{h}_{k-1}$ is derived using the parametric form of $\mathbf{h}_k = \mathbf{f}_k(\mathbf{h}_{k-1})$.



- We need to use transformations $\mathbf{h}_k = \mathbf{f}_k(\mathbf{h}_{k-1})$ for which we can easily compute log-determinant of the Jacobian matrix $\log|\det(\partial \mathbf{h}_k / \partial \mathbf{h}_{k-1})|$.

Real NVP

(Dinh et al., 2016)

Coupling layer

- Suppose we have two variables x_1, x_2 and a function that maps $x = (x_1, x_2)$ to $y = (y_1, y_2)$:

$$y_1 = x_1$$
$$y_2 = g(x_2, m(x_1))$$

  where g is an invertible map with respect to its first argument given the second one, for example:

$$g(a, b) = a + b$$
$$g(a, b) = ab,\ b \neq 0$$

- This mapping is bijective and we can invert the mapping using:

$$x_1 = y_1$$
$$x_2 = g^{-1}(y_2; m(y_1))$$

Affine coupling layer

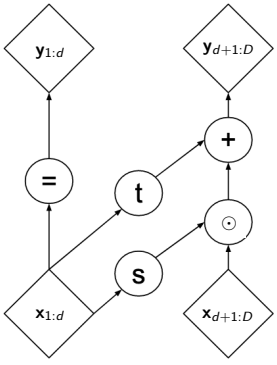
- An invertible transformation with two inputs and outputs:

$$y_1 = x_1$$
$$y_2 = g(x_2, m(x_1)), \quad g \text{ is invertible wrt } x_2$$

- We can generalize this idea to vectors $\mathbf{x}$. We can split a vector $\mathbf{x}$ into two halves $(\mathbf{x}_{1:d}, \mathbf{x}_{d+1:D})$ and apply the following transformation:

$$\mathbf{y}_{1:d} = \mathbf{x}_{1:d}$$
$$\mathbf{y}_{d+1:D} = g(\mathbf{x}_{d+1:D}, \mathbf{x}_{1:d}) = \mathbf{x}_{d+1:D} \odot \exp(s(\mathbf{x}_{1:d})) + t(\mathbf{x}_{1:d})$$

  - s and t are functions $\mathbb{R}^d \mapsto \mathbb{R}^{D-d}$
  - $\odot$ is the Hadamard product or element-wise product



Forward propagation

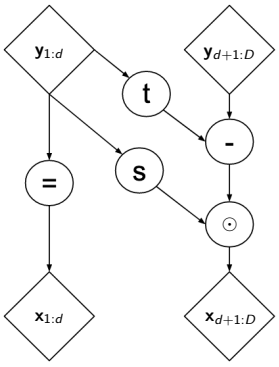
Inverting the affine coupling layer

- Forward propagation

$$\mathbf{y}_{1:d} = \mathbf{x}_{1:d}$$
$$\mathbf{y}_{d+1:D} = \mathbf{x}_{d+1:D} \odot \exp(s(\mathbf{x}_{1:d})) + t(\mathbf{x}_{1:d})$$

- In order to generate samples from the model, we need to invert the transformation. We do this with inverse propagation through the layer:

$$\mathbf{x}_{1:d} = \mathbf{y}_{1:d}$$
$$\mathbf{x}_{d+1:D} = (\mathbf{y}_{d+1:D} - t(\mathbf{y}_{1:d})) \odot \exp(-s(\mathbf{y}_{1:d}))$$



Inverse propagation

Affine coupling layer: Computation of the Jacobian determinant

- The Jacobian of this transformation is

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}}^{\top} = \begin{bmatrix} \mathbf{I}_d & 0 \\ \frac{\partial \mathbf{y}_{d+1:D}}{\partial \mathbf{x}_{1:d}^{T}} & \text{diag}(\exp[s(\mathbf{x}_{1:d})]) \end{bmatrix}$$

- Because the Jacobian is triangular, we can efficiently compute its determinant as

$$\det \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \exp \sum_j s(\mathbf{x}_{1:d})_j$$

- Since computing the Jacobian determinant does not involve computing the Jacobian of s or t, those functions can be arbitrarily complex. Dinh et al. (2016) model s and t as deep convolutional neural networks, whose hidden layers can have more features than their input and output layers.

Affine coupling layer: Partitioning

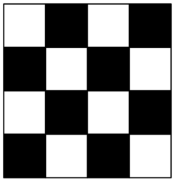
- To apply invertible transformation

$$\mathbf{y}_{1:d} = \mathbf{x}_{1:d}$$
$$\mathbf{y}_{d+1:D} = \mathbf{x}_{d+1:D} \odot \exp(s(\mathbf{x}_{1:d})) + t(\mathbf{x}_{1:d})$$

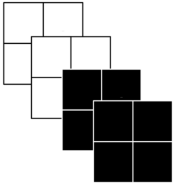
  we need to partition input $\mathbf{x}$ into two parts $\mathbf{x}_{1:d}$ and $\mathbf{x}_{d+1:D}$.

- Real NVP uses two ways of partitioning: checkerboard pattern and channel-wise partitioning (in the figure: either black or white elements remain unchanged).
- Partitioning is implemented using a binary mask $\mathbf{b}$:

$$\mathbf{y} = \mathbf{b} \odot \mathbf{x} + (1 - \mathbf{b}) \odot (\mathbf{x} \odot \exp(\mathbf{s}(\mathbf{b} \odot \mathbf{x})) + t(\mathbf{b} \odot \mathbf{x}))$$



checkerboard pattern



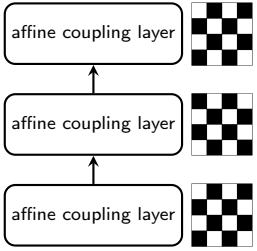
channel-wise partitioning

Combining coupling layers

- Problem with partitioning: the forward transformation leaves components $\mathbf{x}_{1:d}$ unchanged:

$$\mathbf{y}_{1:d} = \mathbf{x}_{1:d}$$
$$\mathbf{y}_{d+1:D} = \mathbf{x}_{d+1:D} \odot \exp(s(\mathbf{x}_{1:d})) + t(\mathbf{x}_{1:d})$$

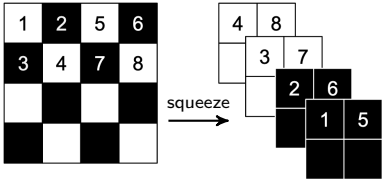
- This difficulty can be overcome by composing coupling layers in an alternating pattern, such that the components that are left unchanged in one coupling layer are updated in the next.



RealNVP: alternating partitioning patterns in a stack of affine coupling layers

Squeeze operation

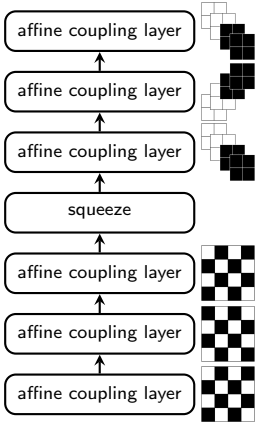
- We often want to reduce the image resolution (e.g., by using strides in convolutional layers).
- In Real NVP, we use the *squeeze* operation for that.
  - We keep the total number of variables same (we need to preserve invertibility).
  - We reduce the spatial size but increase the number of channels.
- The squeeze operation transforms an $s \times s \times c$ tensor into an $\frac{s}{2} \times \frac{s}{2} \times 4c$. For each channel, it divides the image into subsquares of shape $2 \times 2 \times c$, then reshapes them into subsquares of shape $1 \times 1 \times 4c$.



- For better mixing of the variables:
  - The checkerboard pattern is used right before the squeeze operation.
  - Channel-wise partitioning is used right after the squeeze operation.

Multi-scale architecture

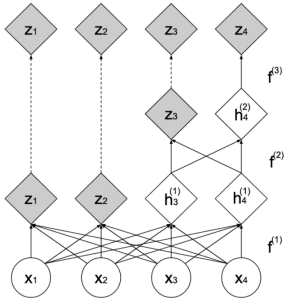
- At each scale, we combine several operations into a sequence:
  - three coupling layers with alternating checkerboard masks
  - a squeezing operation
  - three more coupling layers with alternating channel-wise masking.



One block of Real NVP

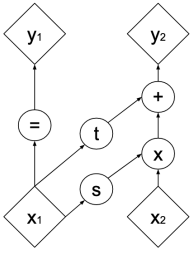
Split operation: Factoring out variables

- For a $n \times n$ image with c channels, the number of dimensions in the input is $n^2 \times c$.
- If we propagate all the $n^2 \times c$ dimensions through all the layers:
  - high computational and memory cost
  - large number of parameters
- The workaround is to apply the *split* operation:
  - Half of the dimensions are directly passed to the output of the network and modeled as Gaussian.
  - The rest of the dimensions are fed to the next layer.
- The purpose is somewhat similar to using pooling layers in standard convolutional networks.



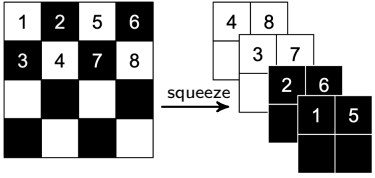
At each step, half the variables are directly modeled as Gaussians, while the other half undergo further transformation.

Real NVP summary: Three types of blocks



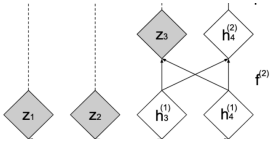
Affine coupling layer:

- Keeps the same dimensions.
- Uses either checkerboard or channel-wise mixing patterns.



Squeeze operation:

- Reduces the spatial resolution by 2 in each dimension.
- Increases the number of channels by 4.



Split operation:

- Removes half of the variables from further computations.

Real NVP: Training

- Forward computation:
  - Compute $\mathbf{z} = \mathbf{f}(\mathbf{x})$
  - Compute the Jacobian determinant and the loss

$$\log p_{\theta}(\mathbf{x}) = \log p_{\theta}(\mathbf{z}) + \sum_{i=1}^{K} \log \left| \det \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \right|$$

    Recall that the determinant was trivial to compute for the affine coupling layer:

$$\det \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \exp \sum_{j} s(\mathbf{x}_{1:d})_j$$

- Backward pass: compute the gradient of the loss wrt parameters $\boldsymbol{\theta}$ of the layers.

Generating samples

- Generating samples is trivial:
  - Generate $\mathbf{z}$ from Gaussian distribution:

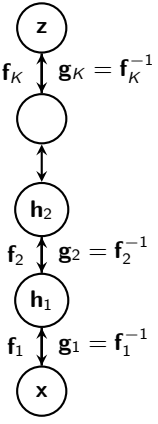
$$\mathbf{z} \sim \mathcal{N}(\mathbf{z}; 0, \mathbf{I})$$

  - Propagate $\mathbf{z}$ through the inverse of $\mathbf{f}$:

$$\mathbf{x} = \mathbf{g}_\theta(\mathbf{z}) = \mathbf{f}^{-1}(\mathbf{z})$$

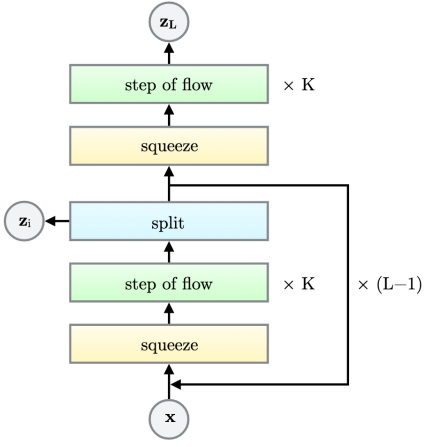
  - We simply need to invert each layer starting from the topmost one:

$$\mathbf{f}^{-1} = \mathbf{f}_1^{-1} \circ \mathbf{f}_2^{-1} \circ \ldots \circ \mathbf{f}_K^{-1}$$



Glow (Kingma and Dhariwal, 2018)

- Glow is further development of flow-based generative models.
- To a great extent, Glow follows the multi-scale architecture introduced in Real NVP.
- They introduce a novel "step of flow" block which is a stack of three layers:
  - Actnorm layer (new layer)
  - Invertible 1×1 convolution (new layer)
  - Affine coupling layer (same as in Real NVP)



Multi-scale architecture of Glow

Samples generated with Glow