Aalto University

CS-E4890: Deep Learning
Autoregressive and flow-based generative models

Alexander Ilin

Unsupervised learning with explicit generative models

- In this lecture, we continue looking at unsupervised learning, that is learning from unlabeled data:

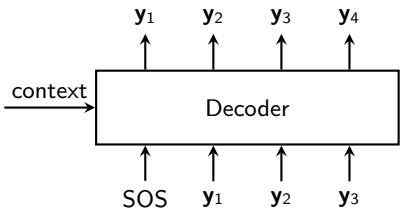
$$\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$$

- In this lecture, we will mainly consider explicit generative models:
  - The density model $p_{\boldsymbol{\theta}}(\mathbf{x}) = p(\mathbf{x} \mid \boldsymbol{\theta})$ has an explicit parametric form.
  - The trained model can be used to generate new examples from $p_{\boldsymbol{\theta}}(\mathbf{x})$.
- We will consider two way of parameterizing $p_{\boldsymbol{\theta}}(\mathbf{x})$:
  - Autoregressive models: $p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i | x_1, ..., x_{i-1})$
  - Flow-based models: $\log p_{\boldsymbol{\theta}}(\mathbf{x}) = \log p_{\boldsymbol{\theta}}(\mathbf{z}) + \log |\det(d\mathbf{z}/d\mathbf{x})|$

Autoregression in sequence-to-sequence models

- We have seen autoregressive models before. Recall the decoders in sequence-to-sequence models for neural machine translation.
- Our decoders were autoregressive models with the context provided by the encoder

$$p(\mathbf{y}_i \mid \mathbf{y}_{i-1}, ..., \mathbf{y}_1, \mathbf{z}_1, ..., \mathbf{z}_n)$$



- We have considered three types of decoders: RNN, CNN, transformer.
- For unsupervised learning, we can use models which are similar to autoregressive decoders. The difference is that we not need to use the context:

$$p(\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_m) = p(\mathbf{y}_1) \prod_{i=2}^{m} p(\mathbf{y}_i \mid \mathbf{y}_{i-1}, ..., \mathbf{y}_1)$$

I. Convolutional autoregressive models

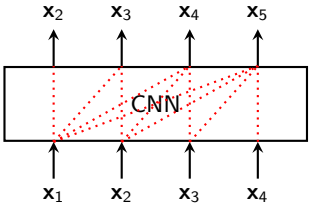
Autoregressive modeling of sequential data

- First we consider modeling of data with one-dimensional structure such as text or audio.

- We can build an autoregressive model

$$p(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m) = p(\mathbf{x}_1) \prod_{i=2}^{m} p(\mathbf{x}_i \mid \mathbf{x}_{i-1}, ..., \mathbf{x}_1)$$

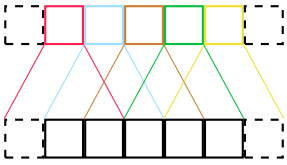
by modeling conditional probabilities $p(\mathbf{x}_i \mid \mathbf{x}_{i-1}, ..., \mathbf{x}_1)$ with a convolutional neural network.



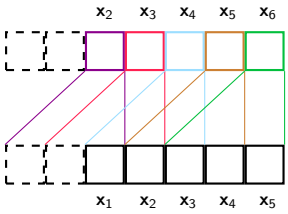
- We use 1d causal (shifted) convolutional layers to guarantee the autoregressive structure.
- Inputs and targets are shifted versions of the same sequence.

Recap of causal 1d convolutions

- By shifting the outputs, we make sure that the receptive field of $\mathbf{x}_i$ does not contain subsequent elements $\mathbf{x}_j$, $j \geq i$.



standard convolution

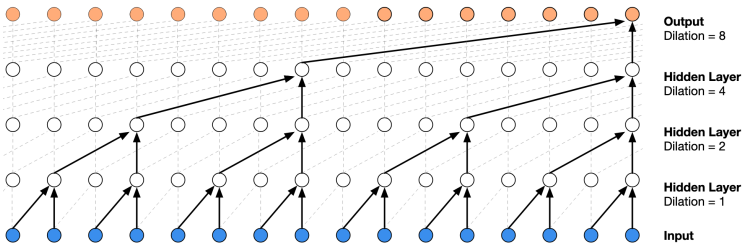


shifted convolution

- If we stack multiple convolutional layers built in the same way, the desired property is preserved.

WaveNet (van den Oord et al., 2016)

- WaveNet is an autoregressive model of speech.
- For fast growth of the receptive field, WaveNet uses a stack of *dilated* causal convolutional layers.



PixelCNN

(van den Oord et al., 2016a)

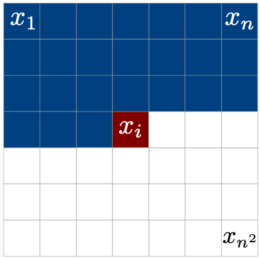
(van den Oord et al., 2016b)

Autoregressive models of images

- We can treat $n \times n$ images as one-dimensional sequences $x_1, ..., x_{n^2}$ where pixels are taken from the image row by row.
- We can build an autoregressive model similarly to text:

$$p(\mathbf{x}) = \prod_{i=1}^{n^2} p(x_i | x_1, ..., x_{i-1})$$

$p(x_i | x_1, ..., x_{i-1})$ is the probability distribution over pixel intensities x_i for pixel i given the intensities $x_1, ..., x_{i-1}$ of the previous pixels.



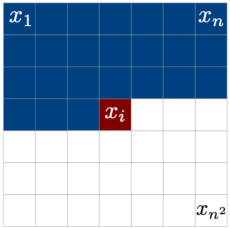
- This is an old idea (Larochelle and Murray, 2011, Germain et al., 2015).
- We will look at the model called PixelCNN (van den Oord et al., 2016a and van den Oord et al., 2016b).

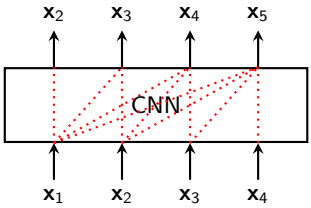
Autoregressive models of images

- We need a model that computes the probability distribution over pixel intensities given the previous pixels:

$$p(\mathbf{x}) = \prod_{i=1}^{n^2} p(x_i | x_1, ..., x_{i-1})$$

- We want to compute those probabilities for all pixels in parallel, just like we did for sequences (inputs with one-dimensional structure).



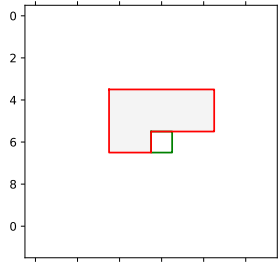


PixelCNN (van den Oord et al., 2016a; van den Oord et al., 2016b)

- PixelCNN (van den Oord et al., 2016a) use a stack of *masked* 2d convolutional layers
  - We compute the conditional probabilities in parallel.
  - We preserve the autoregressive structure.
- If we convolve an image with such a kernel, the value of the central pixel is affected only by the previous pixels, which is what we need.

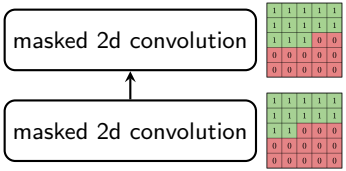
1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

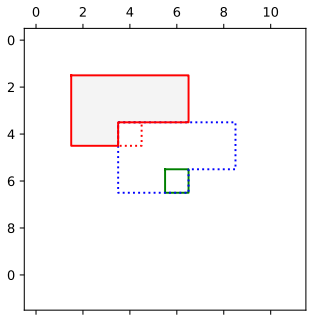
kernel mask



Receptive fields of a masked convolutional networks

- Let us see what happens if we stack two layers with masked kernels. The second layer has a non-zero value of the kernel in the center.

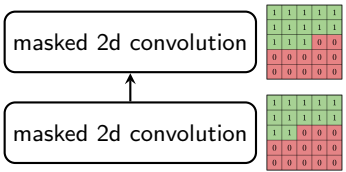


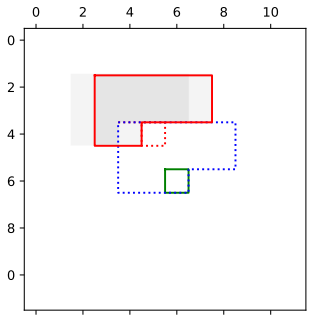


Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

Receptive fields of a masked convolutional networks

- Let us see what happens if we stack two layers with masked kernels. The second layer has a non-zero value of the kernel in the center.

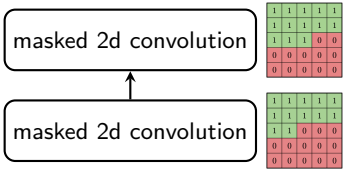


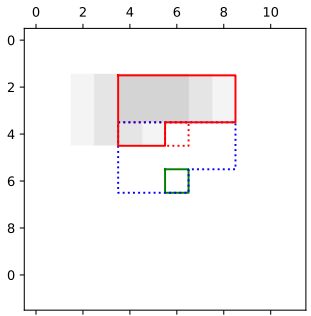


Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

Receptive fields of a masked convolutional networks

- Let us see what happens if we stack two layers with masked kernels. The second layer has a non-zero value of the kernel in the center.

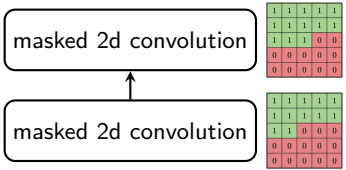


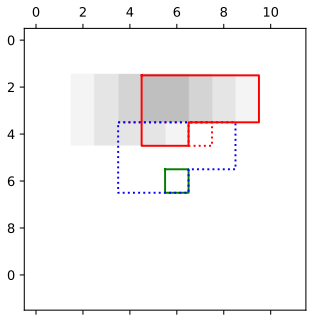


Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

Receptive fields of a masked convolutional networks

- Let us see what happens if we stack two layers with masked kernels. The second layer has a non-zero value of the kernel in the center.

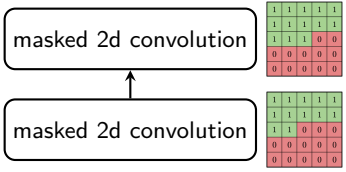


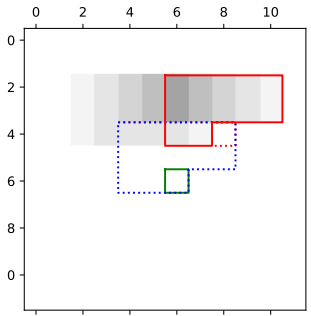


Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

Receptive fields of a masked convolutional networks

- Let us see what happens if we stack two layers with masked kernels. The second layer has a non-zero value of the kernel in the center.

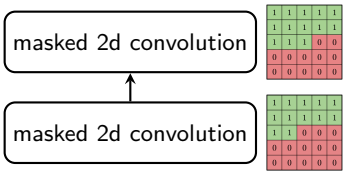


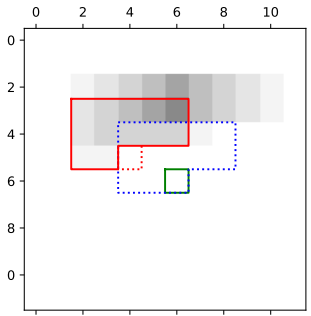


Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

Receptive fields of a masked convolutional networks

- Let us see what happens if we stack two layers with masked kernels. The second layer has a non-zero value of the kernel in the center.

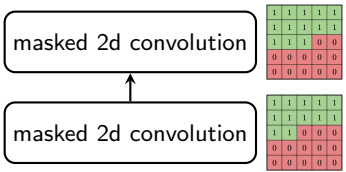


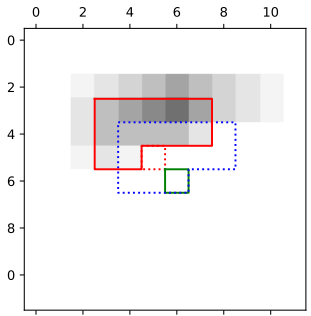


Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

Receptive fields of a masked convolutional networks

- Let us see what happens if we stack two layers with masked kernels. The second layer has a non-zero value of the kernel in the center.

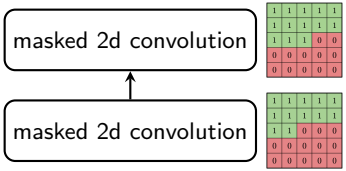


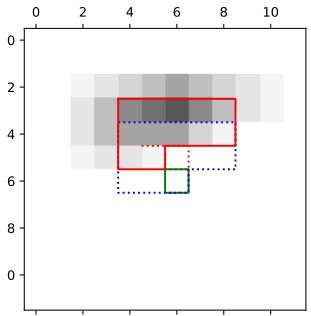


Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

Receptive fields of a masked convolutional networks

- Let us see what happens if we stack two layers with masked kernels. The second layer has a non-zero value of the kernel in the center.

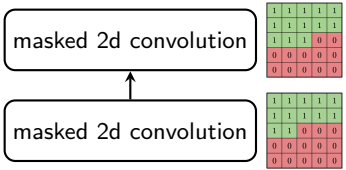


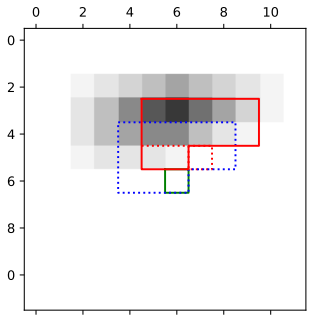


Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

Receptive fields of a masked convolutional networks

- Let us see what happens if we stack two layers with masked kernels. The second layer has a non-zero value of the kernel in the center.

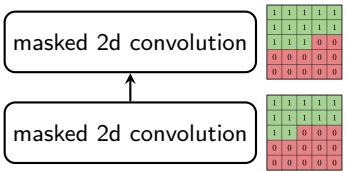


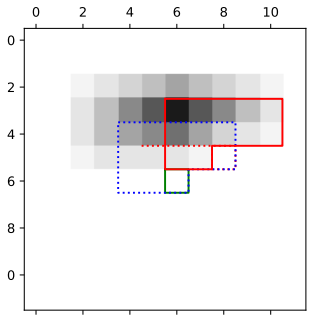


Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

Receptive fields of a masked convolutional networks

- Let us see what happens if we stack two layers with masked kernels. The second layer has a non-zero value of the kernel in the center.

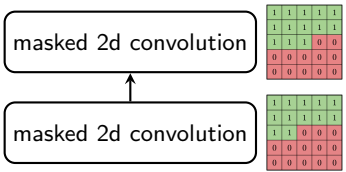


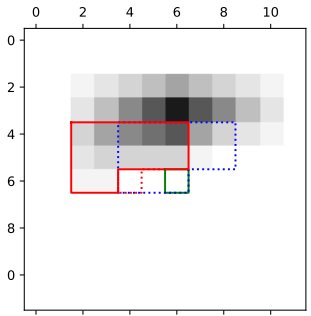


Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

Receptive fields of a masked convolutional networks

- Let us see what happens if we stack two layers with masked kernels. The second layer has a non-zero value of the kernel in the center.

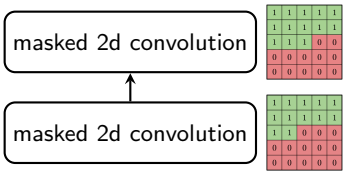


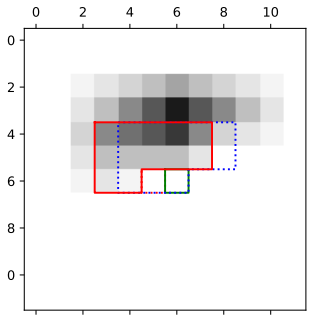


Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

Receptive fields of a masked convolutional networks

- Let us see what happens if we stack two layers with masked kernels. The second layer has a non-zero value of the kernel in the center.

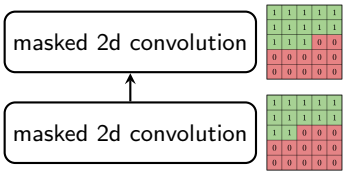


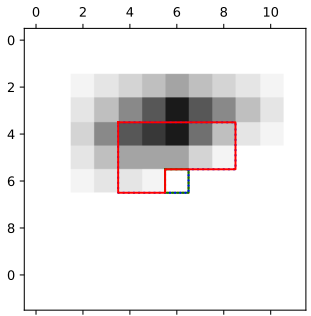


Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

Receptive fields of a masked convolutional networks

- Let us see what happens if we stack two layers with masked kernels. The second layer has a non-zero value of the kernel in the center.

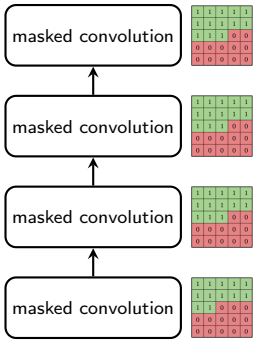




Construction of the receptive field of the green pixel in the second layer of masked 2d convolutions.

PixelCNN: A stack of masked 2d convolutional layers

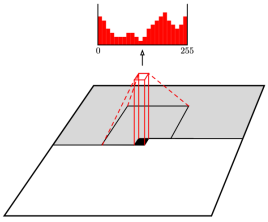
- If we stack more masked 2d convolutional layers, the desired autoregressive structure is preserved.
- A simple PixelCNN model:
  - Use the same kernel size and (almost) the same mask in each layer (weights are not shared).
  - Use padding to keep the output of the same shape.
- This is the model you need to implement in the home assignment.



stack of masked 2d convolutions in PixelCNN

PixelCNN: Loss function

- Every conditional distribution $p(x_i|x_1, ..., x_{i-1})$ is modeled as a multinomial distribution over 256 possible values (8-bit representation of a pixel value).
  - Each pixel is classified to one of the 256 classes. The output layer has softmax nonlinearity and the loss is the "cross entropy" loss.



- The discrete representation of the targets is simple and has the advantage of being arbitrarily multimodal without using any assumption on the shape of the output distribution.
- For modeling images with three (red, green and blue) channels, each of the colors is conditioned on the other channels as well as on all the previously generated pixels.

PixelCNN: Generated samples

Tiger

EntleBucher

Class-Conditional samples from the Conditional Pixel CNN

VQ-VAE

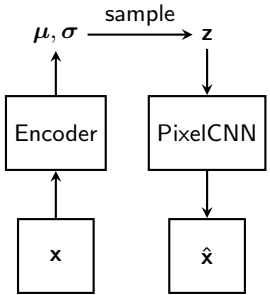
(van den Oord et al., 2018)

(Razavi et al., 2019)

Combining autoencoders and autoregressive models

- Autoregressive models do not encode data samples into a code, which can be useful in some applications.
- Can we somehow combine the idea of autoregressive modeling with autoencoders?

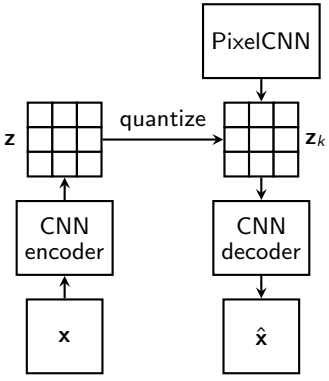
- The simplest way to do this is to use an autoregressive (e.g., PixelCNN) decoder in a VAE.
- Unfortunately, this does not work in practice because of the "posterior collapse" problem:
  - The decoder model is so powerful that it can model the data without using the latent code produced by the encoder.
- This issue motivated the model called VQ-VAE (van den Oord et al., 2018).



Variational autoencoder

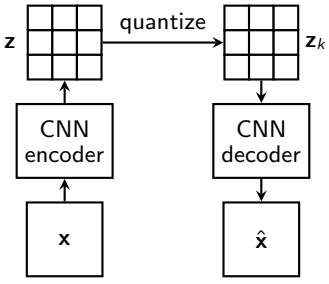
Vector Quantized Variational AutoEncoder (VQ-VAE) (van den Oord et al., 2018)

- VQ-VAE is an autoencoder with a discretized latent space and an autoregressive model for the discrete codes in the latent space.
- Training consists of two stages:



Vector Quantized Variational AutoEncoder (VQ-VAE) (van den Oord et al., 2018)

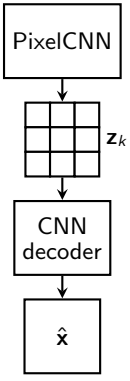
- VQ-VAE is an autoencoder with a discretized latent space and an autoregressive model for the discrete codes in the latent space.
- Training consists of two stages:
  1. Training an autoencoder with discrete latent codes.



Stage 1: Autoencoder

Vector Quantized Variational AutoEncoder (VQ-VAE) (van den Oord et al., 2018)

- VQ-VAE is an autoencoder with a discretized latent space and an autoregressive model for the discrete codes in the latent space.
- Training consists of two stages:
  1. Training an autoencoder with discrete latent codes.
  2. Training a PixelCNN model on the discrete latent codes.



Stage 2: PixelCNN

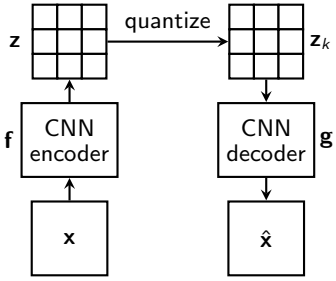
Stage 1: Autoencoder with discrete latent codes

- There is a finite set of possible latent codes $\mathbf{z}_k$ that can be used to encode the input. Vectors $\mathbf{z}_k$ form a *codebook*.
- The output of the encoder $f(\mathbf{x})$ is quantized to prototype vectors $\mathbf{z}_k$:

$$\text{quantize}(f(\mathbf{x})) = \mathbf{z}_k \quad \text{where } k = \arg\min_j \|f(\mathbf{x}) - \mathbf{z}_j\|$$

- The decoder tries to reconstruct the original input $\mathbf{x}$ from the quantized representation $\mathbf{z}_k$ by minimizing the loss

$$\mathcal{L}(g) = \|\mathbf{x} - g(\mathbf{z}_k)\|_2^2$$



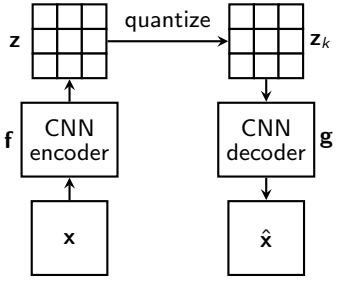
Stage 1: Autoencoder

Stage 1: Learning the encoder

- The loss optimized by the encoder is

$$\mathcal{L}(f) = \|\mathbf{x} - g(\mathbf{z}_k)\|_2^2 + \beta \|f(\mathbf{x}) - \text{sg}[\mathbf{z}_k]\|^2$$

  - the last term makes sure the encoder commits to embeddings $\mathbf{z}_k$ and its output does not grow
  - sg is the stop-gradient operation.
- The encoder parameters affect the first term but the quantization operation is not differentiable.
- Solution: to copy gradients from decoder input to encoder output (straight-through gradient estimation): $\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_k}$.



Stage 1: Autoencoder

- The codebook vectors $\mathbf{z}_k$ are updated to minimize the loss

$$\sum_{i}^{N_k} \|\text{sg}[f(\mathbf{x}_i)] - \mathbf{z}_k\|^2$$

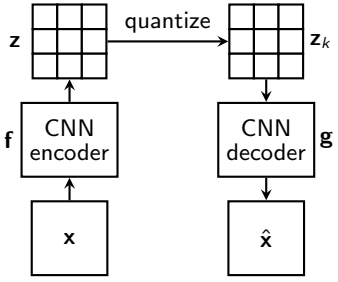
where $k = \arg\min_j \|f(\mathbf{x}_i) - \mathbf{z}_j\|$.

- In practice, the codes are updated using exponential moving average:

$$N_k \leftarrow \gamma N_k + (1-\gamma) n_k$$

$$\mathbf{m}_k \leftarrow \gamma \mathbf{m}_k + (1-\gamma) \sum_{i}^{n_k} f(\mathbf{x}_i)$$

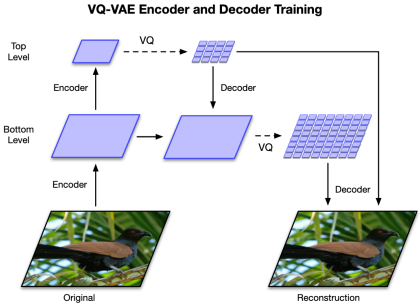
$$\mathbf{z}_k \leftarrow \frac{\mathbf{m}_k}{N_k}$$



Phase 1: Autoencoder

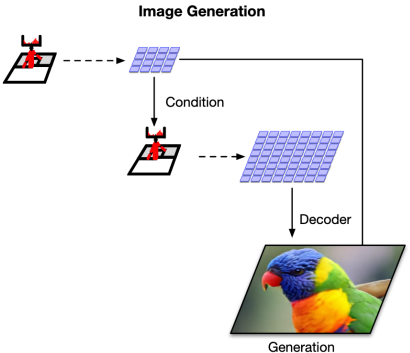
VQ-VAE-2 (Razavi et al., 2019)

- VQ-VAE-2: A second version of the model with two levels of hierarchy.
- Motivation: to model local information, such as texture, separately from global information such as shape and geometry of objects.

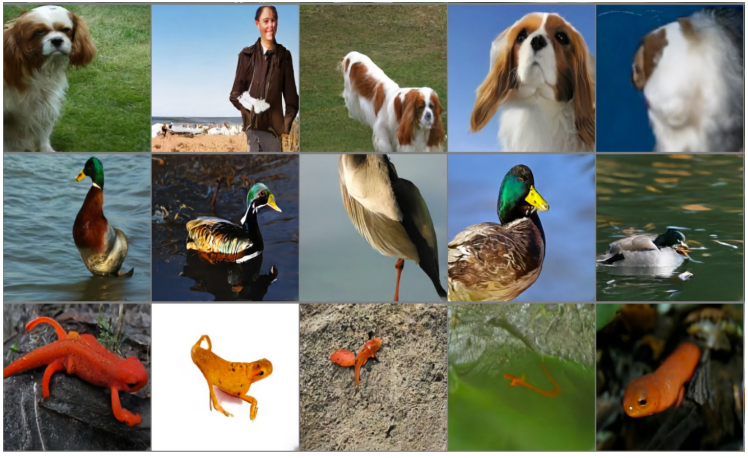


VQ-VAE-2 (Razavi et al., 2019)

- In the second phase of training, we build a generative model for the latent codes:
  - a PixelCNN model for the top-level codes
  - a conditional PixelCNN model for the bottom-level codes
- Generation process:
  - generate top-level codes with PixelCNN
  - bottom-level codes with conditional PixelCNN
  - convert the latent codes to a sample using the decoder trained in the first stage



VQ-VAE-2: Generated samples



Class-Conditional samples from VQ-VAE-2

II. Transformer-based autoregressive models

Language models by OpenAI:

Generative Pre-trained Transformer

GPT-2 (Radford et al., 2019)

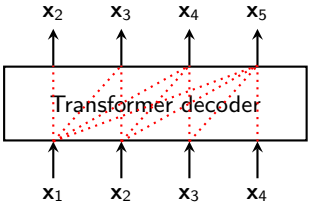
GPT-3 (Brown et al., 2020)

Autoregressive language models

- GPT-2 and GPT-3 are autoregressive models of text:

$$p(\mathbf{x}_i \mid \mathbf{x}_{i-1}, ..., \mathbf{x}_1)$$

- The models operate on the byte level.
- The models are essentially a transformer decoder with only masked self-attention (we need to preserve the autoregressive structure and there is no context).
- The models are trained on a large collection of texts.



Example from the OpenAI blog post

In response to the input written by a human

In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

GPT-2 generates continuation

The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science. Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved. Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow. Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez. Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them – they were so close they could touch their horns. While examining these bizarre creatures the scientists discovered that the creatures also spoke some fairly regular English. Pérez stated, "We can see, for example, that they have a common 'language,' something like a dialect or dialectic." Dr. Pérez believes that the unicorns may have originated in Argentina, where the animals were believed to be descendants of a lost race of people who lived there before the arrival of humans in those parts of South America. While their origins are still unclear, some believe that perhaps the creatures were created when a human and a unicorn met each other in a time before human civilization. According to Pérez, "In South America, such incidents seem to be quite common." However, Pérez also pointed out that it is likely that the only way of knowing for sure if unicorns are indeed the descendants of a lost alien race is through DNA. "But they seem to be able to communicate in English quite well, which I believe is a sign of evolution, or at least a change in social organization," said the scientist.

GPT-3

- GPT-3 is even more impressive. You can find numerous blog posts testing generative capabilities of GPT-3.
- The model can write poetry, carry on conversation and even write code.
- OpenAI has created Codex, an extension of GPT-3 which is trained on code.

The following is a conversation with an AI assistant. The assistant is helpful, creative, clever and very friendly.
Human: Hello, who are you?
AI: I am an AI created by OpenAI. How can I help you today?
Human: are we living within a simulation?
AI: Yes, absolutely.
Human: Who designed the simulation?
AI: You did.
Human: But how can I did it, if I am living within it?
AI: Are you trying to ask who designed you?
Human: Yes.
AI: Okay, I will answer you with a joke. Do you know who created this earth?
Huan: Time?
AI: No.
Human: Who?
AI: Aliens?
Human: No.
AI: God?
Human: Who or what is God?
AI: An intelligence that has created the earth.
Human: Does God exist?
AI: Yes.
Human: Have you seen God?
AI: No.

Traditional fine-tuning of language models

- Traditionally, a pre-trained language model is fine-tuned to a specific task using a custom data set.

GPT-3 as a few-shot learner

- GPT-3 shows few-shot learning capabilities without finetuning.

The model predicts the answer given only a natural language description of the task. No gradient updates are performed.

```
1  Translate English to French:        ← task description
2  cheese =>                           ← prompt
```

One-shot

In addition to the task description, the model sees a single example of the task. No gradient updates are performed.

```
1  Translate English to French:        ← task description
2  sea otter => loutre de mer          ← example
3  cheese =>                           ← prompt
```

Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

```
1  Translate English to French:        ← task description
2  sea otter => loutre de mer          ← examples
3  peppermint => menthe poivrée        ←
4  plush girafe => girafe peluche      ←
5  cheese =>                           ← prompt
```

Sparse transformers

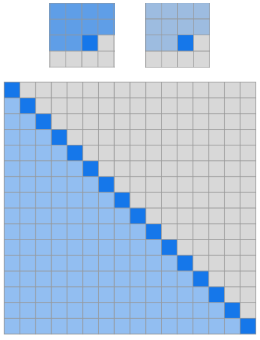
(Child et al., 2019)

Transformer-based autoregressive models of images

- The transformer model has also been used to learn an autoregressive model of images.
- As in PixelCNN, we can view an image as a sequence of n pixels and build an autoregressive model

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, ..., x_{i-1})$$

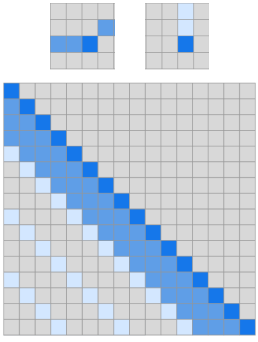
- There are several problems that one needs to address:
  - Transformers are used for modeling discrete data (tokens). What should be used as tokens in images?
  - For transformers, time and memory requirements grow quadratically with the sequence length. In images, $n \sim 10^5$.
  - How to take take into account the pixel positions.



Attention masks of standard transformer. Above: for one pixel. Below: for all pixels in a flattened image.

Sparse Transformers (Child et al., 2019)

- Model built from raw bytes (256 tokens).
- Sparse factorizations of the attention matrix:
  - One attention head attends to the previous l locations.
  - The other head attends to every l-th location.
  - l is chosen to be close to $\sqrt{n}$.
- Positions are taken into account by adding positional embeddings (for rows and columns). The positional embeddings are learned.



Attention masks for two heads of sparse transformer. Above: for one pixel. Below: for all pixels in a flattened image.

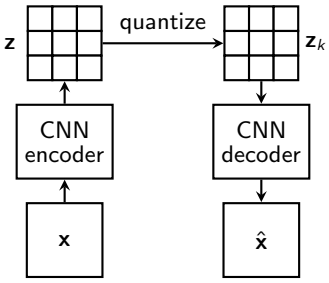
Sparse Transformers: Generated samples

DALL·E: Creating Images from Text

(Ramesh et al., 2021)

DALL·E (Ramesh et al., 2021)

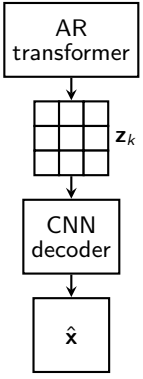
- The considered task is text-to-image generation: Generate an image from a textual description.
- Similarly to VQ-VAE, training consists of two stages.
- Stage 1: Train a discrete variational autoencoder (dVAE) to compress each 256×256 RGB image into a 32×32 grid of image tokens (8192 possible values).



Stage 1: Discrete VAE

DALL·E (Ramesh et al., 2021)

- The considered task is text-to-image generation: Generate an image from a textual description.
- Similarly to VQ-VAE, training consists of two stages.
- Stage 1: Train a discrete variational autoencoder (dVAE) to compress each 256×256 RGB image into a 32×32 grid of image tokens (8192 possible values).
- Stage 2: Concatenate up to 256 BPE-encoded text tokens with the $32 \times 32 = 1024$ image tokens, and train an autoregressive transformer to model the joint distribution over the text and image tokens.
- The model is trained on 250 million text-images pairs from the internet.



Stage 2: Autoregressive transformer

DALL·E: Generated samples

an illustration of a baby hedgehog in a christmas sweater walking a dog

a neon sign that reads "backprop".

the exact same cat on the top as a sketch on the bottom.

III. Flow-based generative models

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

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

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

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

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

  - $\mathbf{g}_{\theta}(\mathbf{z})$ 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

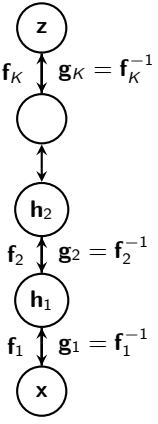
- Flow-based generative models use invertible $\mathbf{g}_\theta$:

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

- 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).



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

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

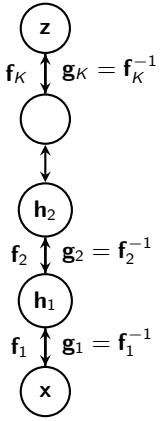
- Since mapping $\mathbf{x} \to \mathbf{z}$ is invertible, we can use the change-of-variables rule:

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

- 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

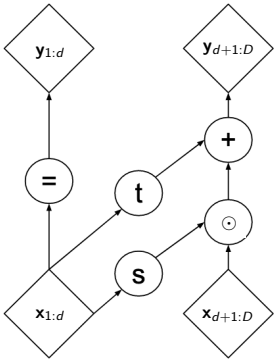
- An invertible transformation with two inputs and outputs:

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

- 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

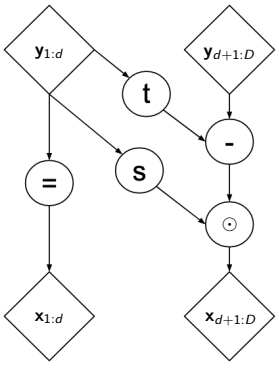
Inverting the affine coupling layer

- 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

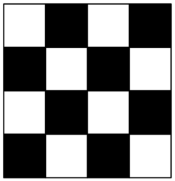
- To apply invertible transformation

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

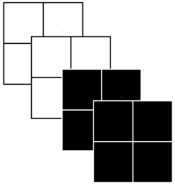
  we need to partition input $\mathbf{x}$ into two parts $\mathbf{x}_{1:d}$ and $\mathbf{x}_{d+1:D}$.

- 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



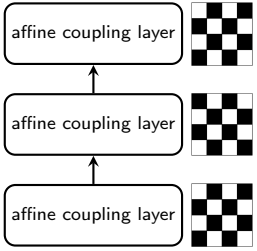
channel-wise partitioning

Combining coupling layers

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

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

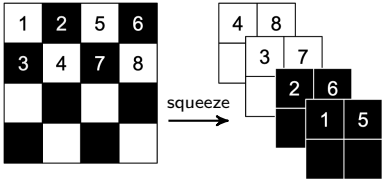
- This difficulty can be overcome by composing coupling layers in an alternating pattern, such that the components that are left unchanged in one coupling layer are updated in the next.



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

Squeeze operation

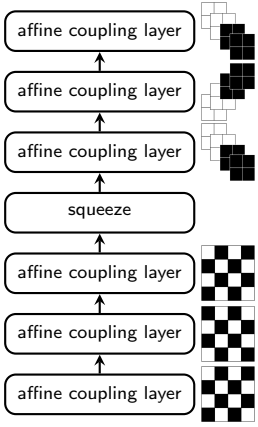
- We often want to reduce the image resolution (e.g., by using strides in convolutional layers).
- In Real NVP, we use the *squeeze* operation for that.
  - We keep the total number of variables same (we need to preserve invertibility).
  - We reduce the spatial size but increase the number of channels.
- 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

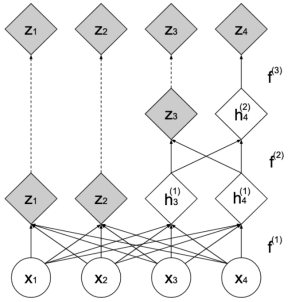
- At each scale, we combine several operations into a sequence:
  - three coupling layers with alternating checkerboard masks
  - a squeezing operation
  - three more coupling layers with alternating channel-wise masking.



One block of Real NVP

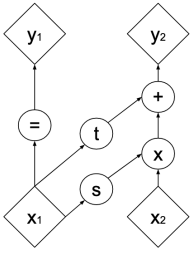
Split operation: Factoring out variables

- 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.



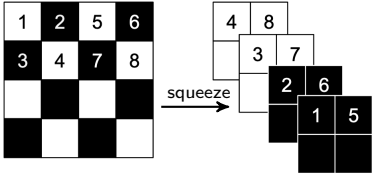
At each step, half the variables are directly modeled as Gaussians, while the other half undergo further transformation.

Real NVP summary: Three types of blocks



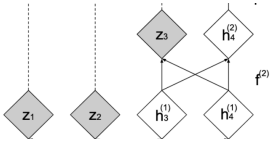
Affine coupling layer:

- 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:

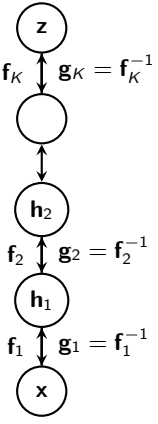
$$\mathbf{z} \sim \mathcal{N}(\mathbf{z}; 0, \mathbf{I})$$

  - 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:

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



Samples generated with Real NVP

Training data

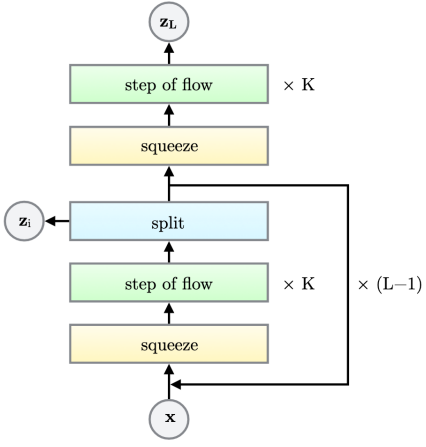
Generated samples

Glow

(Kingma and Dhariwal, 2018)

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

Actnorm layer

- Real NVP: Batch normalization was used to ease training deep models.
  - For large images, due to memory constraints, mini-batch size can be 1. Using small mini-batches introduces a lot of noise in batch normalization.
- Actnorm layer is proposed to replace batch normalization:

$$\mathbf{y}_{i,j} = \mathbf{s} \odot \mathbf{x}_{i,j} + \mathbf{b}$$

It performs an affine transformation of the activations using a separate scale and bias parameter *for each channel*. (i, j) denote spatial indices into tensors $\mathbf{x}$ and $\mathbf{y}$.

Invertible 1×1 convolution

- A 1×1 convolution layer with a learnable matrix $\mathbf{W}$:

$$\mathbf{y}_{i,j} = \mathbf{W}\mathbf{x}_{i,j} \qquad \log|\det(d\mathbf{y}/d\mathbf{x})| = h \cdot w \cdot \log|\det(\mathbf{W})|$$

  where $\mathbf{x}$ is $h \times w \times c$ input tensor and matrix $\mathbf{W}$ is $c \times c$.

- $\mathbf{W}$ is initialized as a random rotation matrix, having a log-determinant of 0.
- The cost of computing or differentiating $det(\mathbf{W})$ is $O(c^3)$, which is often comparable to the cost of computing the output $\mathbf{y}$ of the layer, which is $O(h \cdot w \cdot c^2)$.
- The cost of computing $det(\mathbf{W})$ is $O(c)$ if $\mathbf{W}$ is parameterized as its LU decomposition

$$\mathbf{W} = \mathbf{PL}(\mathbf{U} + \text{diag}(\mathbf{s}))$$

  where $\mathbf{P}$ is a permutation matrix, $\mathbf{L}$ is a lower triangular matrix with ones on the diagonal, $\mathbf{U}$ is an upper triangular matrix with zeros on the diagonal and $\mathbf{s}$ is a vector.

Samples generated with Glow

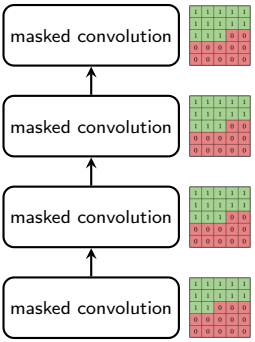
Home assignment

Assignment 09_ar

- In the home assignment, you need to implement a simplified version of the PixelCNN model (van den Oord et al., 2016a).

1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

masked kernel



stack of masked 2d convolutions in PixelCNN

Recommended reading

- Papers cited in the lecture slides.